



NEXT.JS VERCEL COST OPTIMIZATION GUIDE

Jakub Dakowicz
& Chris Lojniewski

Table of contents

Reducing Vercel Hosting Costs	2
Understanding Vercel's Pricing Structure	3
Key Billable Resources	3
Hidden Costs and Common Billing Surprises	4
Analyzing Your Current Costs	5
Using Vercel's Analytics Dashboard for Cost Hotspots	5
Identifying Optimization Opportunities	9
Rendering Strategy Optimization	10
Comparing SSG vs. SSR Cost Implications	10
Implementation Guide for Cost-Efficient Rendering	11
Converting High-Cost SSR Pages to Static Generation	13
Serverless Function Optimization	17
Understanding Serverless Function Costs	17
Function Performance Optimization Techniques	22
Memory Allocation and Execution Duration Management	24
Leveraging Function Concurrency Effectively	25
Dependency Management Best Practices	26
Implementing Effective Caching	27
Bandwidth and Data Transfer Optimization	30
Understanding Bandwidth Costs and Consumption Factors	30
Front-End Asset Optimization	30
API Response Optimization Techniques	33
Prefetching Optimization Strategies	38
Practical Implementation Guide	39
Case Study: Real-World Bandwidth Reduction	40
Vercel Defaults Optimization	41
Fluid Compute for Cost Savings	41
New Image Optimization Pricing Model Strategies	42
What to Do Next	46

Reducing Vercel Hosting Costs

Vercel makes it easy to deploy and host Next.js apps. Once your app starts getting real traffic, however, the hosting bill can grow fast. Even worse, unless you're checking the usage dashboard regularly, it's easy to miss where those extra costs are coming from.

Our guide is made for those who want to understand what they're paying for, and how to bring those costs down without rewriting the app. It covers what drives Vercel costs, how to spot opportunities for optimization using Vercel's built-in tools, and specific ways to improve rendering, serverless functions, and bandwidth usage.

You don't need to change your architecture or jump through multiple hoops to see results. The biggest savings often come from small, targeted changes, and we'll show you how those adjustments can save both money and time.

Understanding Vercel's Pricing Structure

When optimizing your Next.js application's hosting costs on Vercel, it's essential to understand what you're paying for. Vercel's pricing model includes several components that might not be immediately obvious. Knowing these can help you target your optimization efforts where they'll have the biggest impact.

Key Billable Resources

Vercel's billing is primarily based on three resource types that directly impact your monthly costs: bandwidth, function executions, and build minutes.

Bandwidth (Fast Data Transfer)

Bandwidth often represents the largest portion of Vercel costs for growing applications. The Pro plan includes 1TB of bandwidth per month, and additional bandwidth is billed at **\$0.15/GB**. This covers all data transferred between Vercel's infrastructure and your users, including HTML, JavaScript, CSS, images, and API responses

A common surprise for teams is how quickly they can exceed the included bandwidth. For example, a media-rich application with **100,000 monthly users** might easily transfer 3-5TB of data, resulting in **\$300-600** in additional bandwidth charges alone.

Serverless Function Execution

Your Next.js application's dynamic functionality runs on serverless functions, which are billed based on:

- **Execution time:** Measured in GB-seconds (memory allocation × execution time)
- **Invocation count:** Number of times your functions are called
- **Memory allocation:** How much memory each function is allocated

Every API route, server component, and server-side rendered page in your Next.js application consumes these resources. To give you an example, a server-rendered page using the default **1.7GB** memory allocation that takes **300ms** to generate will consume **0.3 GB-seconds** per view.

Build Minutes

Each time you deploy your application, the build process consumes build minutes. Preview deployments for pull requests also consume build minutes, and complex applications with many dependencies can consume a significant amount of them.

While build minutes **typically represent a smaller portion of costs** compared to bandwidth and function execution, they can accumulate quickly for teams with frequent deployments or many preview environments.

Hidden Costs and Common Billing Surprises

Beyond the obvious resources, several factors can lead to unexpected cost increases:

Team Size Impact

The Pro plan is billed at **\$20 per team member per month**. As your development team grows, your base subscription cost increases proportionally. A team of 10 developers results in a **\$200 monthly base cost** before any resource usage.

Add-ons and Supplementary Services

Additional Vercel services can increase your bill:

- Web Analytics (**\$10/month** per project)
- Speed Insights (**\$20/month** per project)
- Image Optimization (now uses a transformation-based billing model)

Although all of them provide value, sometimes they're enabled without full awareness of their cost impact.

Understanding these components of Vercel's pricing structure is the first step toward effective cost optimization. Knowing this, you can now strategically target your optimization efforts on the aspects of your application that contribute most significantly to your hosting costs.

Analyzing Your Current Costs

To lay down the foundation for implementing optimization strategies, you need to trace the current costs of your Vercel hosting. A targeted analysis will help you learn how to prioritize your optimization efforts for maximum impact.

Using Vercel's Analytics Dashboard for Cost Hotspots

All Vercel users receive access to powerful analytics tools that reveal how your application consumes resources. Knowing how to interpret this data correctly is an important step towards effective cost optimization.

Navigating the Usage Dashboard

Inside the **Usage tab**, you'll find metrics that directly impact your bill:

- **Bandwidth consumption:** Shows total data transfer and trends over time
- **Serverless function usage:** Displays execution time and invocation counts
- **ISR Costs:** Tracks how deployment processes consume resources

To access your usage data:

1. Log in to your **Vercel dashboard**
2. Select your **team or personal account**
3. Navigate to the **Usage tab**
4. Select your project from **the dropdown** if you have multiple projects

🕒 Current Billing Cycle
📅 Mar 16, 8:00 - Apr 16, 9:00

^

Overview

Product	Included	
🕒 Fast Origin Transfer	1.19 GB / 100 GB	
🕒 Function Duration	9.9 GB-Hrs / 1,000	
🕒 Function Invocations	6K / 1M	\$0 💰
🕒 ISR Reads	59K / 10M	\$0 💰
🕒 Image Optimization - Source Images (Legacy)	25 / 5,000	\$0 💰

🕒

pagepro-website

🕒

🕒

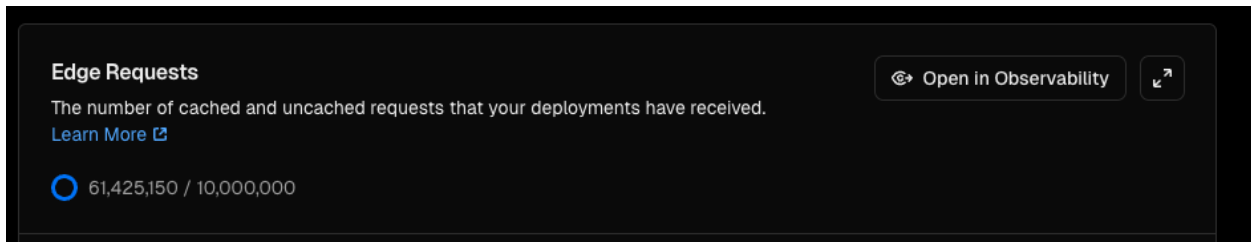
🕒

🕒

🕒

Product	Included	On-demand	Charge
🕒 Speed Insights Data Points	10K / 10K	+4M	\$50.00 💰
🕒 Edge Requests	10M / 10M	+51M	\$126.95 💰
🕒 Edge Middleware Invocations	1M / 1M	+52M	\$34.45 💰
🕒 ISR Writes	2M / 2M	+6.4M	\$33.05 💰
🕒 Fast Origin Transfer	100 GB / 100 GB	+387.96 GB	\$25.51 💰
🕒 ISR Reads	10M / 10M	+22M	\$10.66 💰
🕒 Function Invocations	1M / 1M	+2.9M	\$2.40 💰
🕒 Edge Function Execution Units	1M / 1M	+657K	\$2.00 💰
🕒 Edge Request CPU Duration	1 h / 1 h	+3 m	\$0.03 💰
🕒 Fast Data Transfer	973.28 GB / 1 TB		\$1.85 💰
🕒 Function Duration	285.9 GB-Hrs / 1,000 GB-Hrs		\$0 💰
🕒 Edge Config Reads	0 / 1M		\$0 💰
🕒 Edge Config Writes	0 / 1K		\$0 💰
🕒 Web Analytics Events	0 / 25K		\$0 💰
🕒 Monitoring	0 / 250K		\$0 💰
🕒 Image Optimization - Source Images (Legacy)	0 / 5,000		\$0 💰
🕒 Image Optimization - Transformations	0 / 10K		\$0 💰

- Then open the one you need **in Observability**. In this case, we chose Edge Requests.



You'll find a detailed breakdown of exactly what you're paying for, including how much of your limits you've used and what it's likely to cost you based on current usage. It simplified finding the best opportunities for optimizations. While analyzing, pay extra attention to:

- Resources approaching or exceeding their **included limits**
- Sudden **spikes in usage** that might indicate inefficiencies
- Steady **growth trends** that could lead to higher costs over time

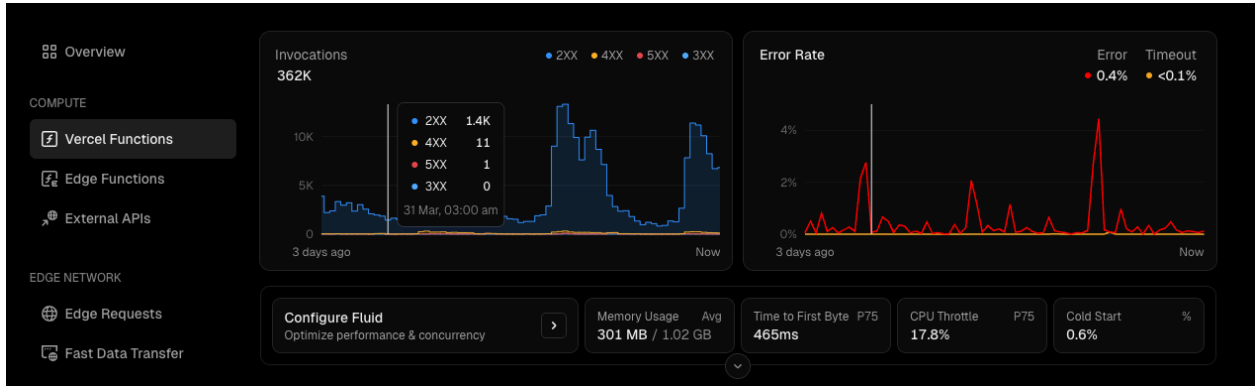
Identifying High-Impact Optimization Targets

To optimize effectively, start by pinpointing which parts of your app are using the most resources. Vercel's **Usage** and **Analytics** dashboards make it easier to identify where to focus your efforts:

- **Top Paths by Bandwidth** - Navigate to your project's Analytics section and sort by bandwidth consumption. This helps you spot pages and assets transferring large amounts of data. Common culprits include image galleries, dashboards with large datasets, product listings, and file or document downloads.
- **Top Functions by Execution Time** - Identify serverless functions that take the longest to run. These are often prime candidates for code performance improvements or memory configuration tweaks.
- **Top Functions by Invocations** - Find out which endpoints are hit most frequently. Small optimizations, like response caching or trimming logic, can have an outsized impact when scaled across millions of requests.

Look for recurring patterns and prioritize the routes or functions with the highest potential payoff. These insights will guide the most impactful optimization decisions.

You can also use the **Vercel Functions dashboard** to spot usage spikes and see which sites or pages triggered them:



For example, here’s a list of the most-used Vercel Function routes, ranked by number of invocations (Top Paths). Switching the view to **GB-hours** highlights the most resource-intensive functions:

Route	Invocations	P75 Duration	GB-Hours	Error Rate
/api/tracking/	85K		2.22	0%
/api/notes	78K		3.02	0%
/api/user	54K		3.53	0%
/api/tracking/	31K		1.25	0%
/api/user/login	19K		2.28	0.1%

And under **Edge Requests**, you can check which paths are consuming the most bandwidth:

Routes	Requests	Cached
/_next/data/[buildId]/en-GB/[...slug].json	1.5M	100%
/_next/data/[buildId]/en/[...slug].json	361K	99.7%
/v2/vitals	245K	0%
/en/[...slug]	213K	94.4%
/api/style	198K	0%

Armed with information provided by the usage dashboard, you can focus your optimization efforts where they'll have the greatest impact.

Identifying Optimization Opportunities

After analyzing your usage patterns, it's time to look for these common inefficiency signals:

Resource Consumption Patterns

Examine your usage data for these typical optimization opportunities:

1. **Server-rendered pages with static content:** Pages that rarely change but use SSR are prime candidates for conversion to static generation
2. **Long-running API routes:** Functions taking over **300ms** to execute often have optimization potential
3. **Routes with high invocation counts:** Frequently called endpoints may benefit from caching
4. **Large bandwidth consumers:** Pages transferring significant data could use asset optimization
5. **Redundant data fetching:** Multiple components or pages fetching the same data

Cost-Benefit Analysis Framework

For each potential optimization, ask yourself a question:

- How much is this specific issue costing monthly?
- How complex would the optimization be?
- Could the change affect functionality or user experience?
- What percentage reduction in resource usage is realistic?

Answering these helps prioritize high-impact, low-effort optimizations that deliver the best return on investment.

For instance, converting a server-rendered product page to static generation with revalidation might take only a few hours of development time but could reduce function execution costs by **90%** for that page and potentially improve performance.

By systematically analyzing your current costs and identifying specific optimization targets, you'll create a focused roadmap that maximizes savings while minimizing development effort. The insights from this analysis will be the basis for all the optimization strategies covered in the next chapters.

Rendering Strategy Optimization

How you render pages in Next.js directly affects your Vercel hosting costs. Smarter rendering choices reduce serverless usage and often lead to better performance.

Comparing SSG vs. SSR Cost Implications

The rendering methods in Next.js come with very different cost profiles:

Static Site Generation (SSG)

SSG generates pages at build time and serves them as static HTML. SSG pages are very cost-efficient because they don't trigger serverless functions for each request. Instead, they're served directly from Vercel's edge network as pre-built HTML.

- **Cost profile:** Minimal runtime costs
- **Resource usage:** Primarily consumes build minutes, not runtime resources
- **Vercel billing impact:** Pages served from CDN with no serverless function execution
- **Cost savings potential:** 80-95% reduction compared to SSR

Server-Side Rendering (SSR)

SSR generates pages on demand for each request. SSR is the most expensive rendering option because every page view triggers function execution. A high-traffic SSR page can easily become the largest component of your Vercel bill.

- **Cost profile:** Highest runtime costs
- **Resource usage:** Every page view executes serverless functions
- **Vercel billing impact:** Consumes both function execution time and invocations
- **Cost implications:** Scales linearly with traffic

Cost Comparison Example

Consider a product page receiving 100,000 views monthly:

Rendering Method	Function Executions	Avg. Duration	Monthly Cost*
SSR	100,000	300ms	~\$30
SSG	0	0	~\$0
SSG with ISR	~500	300ms	~\$0.15

*Based on default 1.7GB memory allocation and approximate Vercel pricing. The accuracy might vary.

The cost difference grows with traffic. Converting high-traffic pages from SSR to SSG (or SSG with revalidation) often provides the single most significant cost-reduction opportunity for Next.js applications on Vercel.

Implementation Guide for Cost-Efficient Rendering

Implementing the optimal rendering strategy requires understanding both the technical approach and the decision framework.

Decision Framework for Rendering Choice

For each page in your application, ask these questions:

- How frequently does the content change?**
 - Static content (rarely changes): Use SSG
 - Semi-dynamic content (changes occasionally): Use SSG with revalidation
 - Highly dynamic content (changes constantly): Use SSR with caching
- Is the content personalized for each user?**
 - No personalization: SSG is ideal
 - Partial personalization: SSG with client-side data fetching for personalized elements
 - Fully personalized: SSR or client-side rendering

3. What are the traffic patterns?

- High-traffic pages benefit most from SSG
- Low-traffic pages have a smaller cost impact regardless of the rendering method

The most cost-effective approach is typically to use SSG for as many pages as possible, adding revalidation or client-side data fetching where needed for dynamic elements.

Implementation in Next.js App Router

To implement static generation in the App Router:

```
// Static generation (default)

export default function Page() {

  return <div>This page uses static generation</div>

}

// With revalidation (ISR approach in App Router)
// /app/pages/[slug]
export const dynamic = 'force-static';
export const revalidate = 3600;

export async function generateStaticParams() {
  const pages = await fetch('https://api.app/pages').then((res) =>
    res.json()
  )
  return pages.map((page) => ({
    slug: String(page.slug),
  })))
}

export default async function Page() {

  const data = await fetch('https://api.example.com/data');

  return <div>{/* Render data */}</div>

}
```

Converting High-Cost SSR Pages to Static Generation

Identifying and converting expensive SSR pages can yield immediate cost savings.

Identifying SSR Pages for Conversion

Run the build command to see how each page is rendered:

```
next build
```

The output shows rendering methods for each route:

- ○ Static routes
- ● Server-rendered routes
- λ API routes

Focus conversion efforts on server-rendered routes with high traffic.

Implementation Strategy for Conversion

For pages initially built with SSR but containing mostly static content:

1. Add revalidation to data fetching:

```
// Before: Dynamic data fetching on every request
const data = await fetch('https://api.example.com/data', { cache: 'no-store' });

// After: Data fetching with revalidation
const data = await fetch('https://api.example.com/data', {
  next: { revalidate: 3600, tags: ["data"] } // 1-hour revalidation
});
```

Common issues

Remember, data fetched with revalidation is cached **across builds**, not just during runtime. This can lead to unexpected stale data unless you properly manage cache invalidation. See: [Next.js Data Cache docs](#)

2. Move user-specific content to client components:

```
// server component with mostly static content
export default async function ProductPage({ params }) {

  // Shared product data (statically generated)

  const product = await getProduct(params.id);

  return (

    <div>

      <ProductDetails product={product} />

      { /* User-specific content in client component */ }

      <UserSpecificContent productId={params.id} />

    </div>

  );
}

// Client component for user-specific content
'use client'

function UserSpecificContent({ productId }) {
  const { data } = useSWR(`/api/user/product/${productId}`, fetcher);
  // Render user-specific content
}
```

3. Convert the site to SSG

```
// Static page

export default async function ProductPage({ params }) {
  const product = await getProduct(params.id);

  return (
    <div>
      <ProductDetails product={product} />
      { /* User-specific content in client component */ }
      <UserSpecificContent productId={params.id} />
    </div>
  );
}

// Thanks to this we now what paths we need to generate and populate
export async function generateStaticParams() {
  const products = await getAllProducts();

  const slugs = products.map(product => ({
    id: product.id,
  }));

  return slugs;
}

export const dynamic = "force-static";

export const dynamicParams = true;
```

4. Implement on-demand revalidation for important updates:

```
// API route to trigger revalidation when content changes

export async function POST(request) {
  const data = await request.json();

  if (data.secret !== process.env.REVALIDATION_SECRET) {
    return Response.json({ error: 'Invalid token' }, { status: 401 });
  }

  try {
```

```
if (data.productId) {
  await revalidatePath(`/products/${data.productId}`);
}

if (data.tagId) {
  revalidateTag(tagId);
}

return Response.json({ revalidated: true });
} catch (err) {
  return Response.json({ error: err.message }, { status: 500 });
}
```

This hybrid approach provides the cost benefits of static generation while maintaining the ability to show fresh content when needed.

Rendering strategy optimization often delivers the largest and most immediate cost savings for Next.js applications on Vercel. By converting high-traffic SSR pages to static generation with appropriate revalidation strategies, you can reduce function execution costs by 80-95% while potentially improving page load performance.

Serverless Function Optimization

Serverless functions power the dynamic capabilities of your Next.js application on Vercel, including API routes, server components, and server-side rendering operations. Optimizing these functions presents one of the most significant opportunities for reducing Vercel hosting costs.

Understanding Serverless Function Costs

Vercel's serverless function pricing model is based on two primary metrics:

Execution Duration and Memory Allocation

Function costs are calculated using GB-seconds, which combines:

- **Execution time:** The duration your function runs (in seconds)
- **Memory allocation:** The amount of memory assigned to your function (in GB)

For example, if your function uses 1GB of memory and runs for 500ms (0.5 seconds), it consumes 0.5 GB-seconds. Reducing either factor directly lowers your costs.

Many Next.js applications unwittingly consume excessive resources because:

- The default memory allocation (1.7GB) is higher than most functions need
- Inefficient code patterns lead to unnecessarily long execution times
- Sequential operations block completion when they could run concurrently

Invocation Frequency Impact

Each function call counts as an invocation, with costs determined by:

- The total number of invocations per month
- How those invocations distribute across your serverless functions

Common patterns that lead to excessive invocations include:

- Overuse of server-side rendering for mostly static content
- Client-side polling that frequently calls API endpoints
- Missing or inadequate caching strategies
- Inefficient data fetching patterns

By addressing both execution efficiency and invocation frequency, you can significantly reduce your serverless function costs.

Function Performance Optimization Techniques

Implement these techniques to improve function performance and reduce execution time:

Code-Level Efficiency

Optimize your function code for performance:

```
// Before: Inefficient database query pattern with N+1 problem
export async function GET(request) {
  const userId = request.nextUrl.searchParams.get('userId');

  // Initial query
  const user = await db.query('SELECT * FROM users WHERE id = $1', [userId]);

  // Separate query for each order (N+1 problem)
  const orders = await db.query('SELECT * FROM orders WHERE user_id = $1', [userId]);

  const enrichedOrders = [];

  for (const order of orders.rows) {
    // Additional query for each order
    const items = await db.query('SELECT * FROM order_items WHERE order_id = $1', [order.id]);

    enrichedOrders.push({
      ...order,
      items: items.rows
    });
  }
}
```

```
});  
  
}  
  
return Response.json({  
  user: user.rows[0],  
  orders: enrichedOrders  
});  
  
}  
  
// After: Optimized query with joins  
export async function GET(request) {  
  const userId = request.nextUrl.searchParams.get('userId');  
  // Single efficient query with joins  
  const result = await db.query(`  
    SELECT  
      u.*,  
      json_agg(  
        json_build_object(  
          'id', o.id,  
          'date', o.date,  
          'items', (  
            SELECT json_agg(oi.*)  
            FROM order_items oi  
            WHERE oi.order_id = o.id  
          )  
      )  
  `);  
}
```

```

    )
  ) as orders

FROM users u

LEFT JOIN orders o ON u.id = o.user_id

WHERE u.id = $1

GROUP BY u.id
`, [userId]);

return Response.json(result.rows[0]);
}

```

This optimization eliminates the N+1 query problem, reducing database operations from potentially dozens to just one, significantly decreasing execution time.

Parallel Data Fetching in Next.js

Next.js makes it possible to fetch data in parallel, reducing total execution time. This can happen automatically in Server Components or be manually implemented in Route Handlers using `Promise.all`.

Server Components: Automatic Parallel Fetching

In Server Components, separate `await` calls are run in parallel by default:

```

// Before: Sequential processing

export async function GET(request) {

  const userId = request.nextUrl.searchParams.get('userId');

  // These queries run sequentially, blocking each other

  const userData = await fetchUserProfile(userId);

  const orderHistory = await fetchOrderHistory(userId);

```

```
const recommendations = await fetchRecommendations(userId);

return Response.json({
  profile: userData,
  orders: orderHistory,
  recommendations: recommendations
});
}

// After: Parallel processing with Promise.all
export async function GET(request) {
  const userId = request.nextUrl.searchParams.get('userId');

  // Run all queries concurrently

  const [userData, orderHistory, recommendations] = await Promise.all([
    fetchUserProfile(userId),
    fetchOrderHistory(userId),
    fetchRecommendations(userId)
  ]);

  return Response.json({
    profile: userData,
    orders: orderHistory,
    recommendations: recommendations
  });
}
```

This reduces wait time automatically by initiating all fetches together and awaiting their resolution.

Route Handlers: Manual Parallel Fetching with `Promise.all`

In Route Handlers, requests made sequentially will block each other. To run them in parallel, you'll need to wrap them in `Promise.all`:

```
// Server Component with automatic parallel data fetching

export default async function ProductPage({ params }) {

  // These requests run in parallel automatically

  const product = await fetchProduct(params.id);

  const reviews = await fetchReviews(params.id);

  const relatedProducts = await fetchRelatedProducts(params.id);

  return (

    <div>

      <ProductDetails product={product} />

      <ReviewSection reviews={reviews} />

      <RelatedProducts products={relatedProducts} />

    </div>

  );

}
```

This pattern can reduce execution time from 50% to 70%, especially when dealing with multiple slow or independent data sources.

Early Return Patterns

Implement early returns to prevent unnecessary processing:

```
export async function GET(request) {  
  
  // 1. Validate request parameters early  
  
  const productId = request.nextUrl.searchParams.get('id');  
  
  if (!productId) {  
  
    return Response.json({ error: 'Missing product ID' }, { status: 400 });  
  
  }  
  
  // 2. Check cache first  
  
  const cachedProduct = await getFromCache(`product:${productId}`);  
  
  if (cachedProduct) {  
  
    return Response.json(cachedProduct);  
  
  }  
  
  // 3. Only perform expensive operations if necessary  
  const product = await fetchProductDetails(productId);  
  
  // 4. Cache result before returning  
  
  await setInCache(`product:${productId}`, product, 3600); // 1 hour TTL  
  
  return Response.json(product);  
  
}
```

Early returns prevent unnecessary computations, reducing both execution time and resource usage.

Memory Allocation and Execution Duration Management

Fine-tuning memory and timeout settings can significantly reduce serverless costs without hurting reliability.

Adjusting Memory Allocation

By default, Vercel allocates 1.7GB of memory to functions, which is more than most need. Dropping that to 256MB or 512MB can cut costs by up to **85%** without impacting performance for typical workloads.

```
{
  "functions": {
    "api/simple/*.js": { "memory": 256 },
    "api/data-processing/*.js": { "memory": 512 }
  }
}
```

Recommended memory by function type:

- Simple API routes: 128–256MB
- Basic data operations: 256–512MB
- Complex logic: 512MB–1GB
- Data-heavy tasks: 1GB+

Always test memory settings under realistic loads to find the right balance between performance and cost.

Managing CPU and Duration

CPU scales with memory, so lower allocations reduce both compute and cost. You can also cap how long a function is allowed to run:

```
{
  "functions": {
    "api/*.js": {
      "memory": 256,
      "maxDuration": 10
    },
  },
}
```

```
"api/reports/*.js": {  
  "memory": 512,  
  "maxDuration": 60  
}  
}  
}
```

Suggested timeout ranges:

- User-facing APIs: **5–10** seconds
- Background jobs: up to **30–60** seconds

Setting sensible `maxDuration` values avoids runaway functions and encourages proper error handling.

Leveraging Function Concurrency Effectively

Implement concurrency patterns to maximize throughput while minimizing execution time:

Batch Processing for Large Datasets

When processing large datasets, use batching to optimize performance:

```
export async function POST(request) {  
  
  const { items } = await request.json();  
  
  // Process in batches instead of all at once or one by one  
  
  const batchSize = 25;  
  
  const results = [];  
  
  for (let i = 0; i < items.length; i += batchSize) {  
  
    const batch = items.slice(i, i + batchSize);  
  
    // Process current batch in parallel  
  
    const batchResults = await Promise.all(  

```

```

    batch.map(item => processItem(item))
  );
  results.push(...batchResults);
}
return Response.json({ results });
}

```

This approach balances parallel execution benefits with resource constraints, preventing memory issues while maintaining good performance.

Dependency Management Best Practices

Dependencies significantly impact function size, cold start times, and memory usage:

Alternative Lightweight Libraries

Replace heavy dependencies with lightweight alternatives:

Heavy Library	Lightweight Alternative	Size Reduction
Moment.js (329KB)	date-fns (86KB)	74%
Lodash (533KB)	lodash-es (individual imports)	Up to 90%
Axios (138KB)	ky (14KB)	90%

For example:

```

// Before: Using the entire lodash library
import _ from 'lodash';

const uniqueItems = _.uniq(items);

```

```
// After: Using direct imports from lodash-es

import { uniq } from 'lodash-es';

const uniqueItems = uniq(items);

// Even better: Using native JavaScript

const uniqueItems = [...new Set(items)];
```

These changes can significantly reduce bundle size and improve cold start times.

Tree-Shaking Friendly Import Patterns

Adopt import patterns that enable effective tree-shaking:

```
// Bad: Imports entire library

import * as utils from '../utils';

// Good: Import only what you need

import { formatDate, calculateTotal } from '../utils';
```

For Next.js API routes and server components, tree-shaking is particularly important as it directly impacts function size and cold start times.

Implementing Effective Caching

Caching reduces both function execution time and invocation frequency:

Response Caching

Implement appropriate caching for API responses:

```
export async function GET(request) {

  // Set cache headers for edge caching
```

```
const response = Response.json(data);

// Cache for 1 hour at the edge, but revalidate if requested
response.headers.set('Cache-Control', 's-maxage=3600, stale-while-revalidate');

return response;
}
```

Data-Level Caching

Implement caching for expensive database queries or external API calls:

```
async function fetchProductWithCache(id) {

  const cacheKey = `product:${id}`;

  // Try to get from cache first

  const cached = await getFromCache(cacheKey);

  if (cached) return cached;

  // If not in cache, fetch from database

  const product = await db.query('SELECT * FROM products WHERE id = $1', [id]);

  // Store in cache with TTL

  await setInCache(cacheKey, product, 300); // 5 minutes

  return product;
}
```

Or use Next.js unstable cache:

```
async function unstable_cache(async fetchProductWithCache(id) {  
  
  const product = await db.query('SELECT * FROM products WHERE id = $1', [id]);  
  return product;  
  
}, ["fetchProduct"])
```

Effective caching can reduce function execution time by **50-99%** for frequently accessed data while also reducing the number of function invocations.

Keep in mind: in high-traffic apps, even small wins add up. **Shaving off just 100ms per function call can lead to serious savings when you're dealing with millions of requests.** Optimizing serverless functions is one of the most effective ways to cut down Vercel hosting costs.

The techniques covered in this chapter can help you slash function costs by **40-80%**, often while speeding up your app.

Bandwidth and Data Transfer Optimization

Bandwidth costs often represent the largest component of Vercel hosting expenses for growing Next.js applications. As your traffic increases, these costs can escalate rapidly, making bandwidth optimization critical for managing your overall hosting budget.

Understanding Bandwidth Costs and Consumption Factors

Vercel's bandwidth billing follows a straightforward but comprehensive model:

- **Pro plan allowance:** 1TB of bandwidth included monthly, with additional usage billed at \$0.15/GB
- **Metered usage:** All data transferred between Vercel's infrastructure and your users counts toward your bandwidth consumption

This includes all content delivered to users:

- HTML, CSS, and JavaScript files
- Images, videos, and other media
- API responses and data payloads
- WebSocket data

For content-heavy applications, media files often account for 60-80% of bandwidth. For data-driven applications, API responses might be the primary factor.

Front-End Asset Optimization

JavaScript Bundle Optimization Techniques

JavaScript often represents the largest portion of code-related bandwidth consumption:

```
// Code splitting with dynamic imports

const LargeComponent = dynamic(() => import('./LargeComponent'), {
  loading: () => <p>Loading...</p>
});

// Import only what you need

import { formatDate, formatCurrency } from '../utils';

// Instead of: import * as utils from '../utils';
```

Analyze your bundle size to identify optimization opportunities:

```
# Install bundle analyzer

yarn add -D @next/bundle-analyzer

# Configure in next.config.js

const withBundleAnalyzer = require('@next/bundle-analyzer')({
  enabled: process.env.ANALYZE === 'true',
});

module.exports = withBundleAnalyzer({
  // Your Next.js config
});

# Run analysis

ANALYZE=true yarn build
```

These techniques can reduce JavaScript bundle sizes by **30-50%**, directly lowering bandwidth consumption.

CSS and Font Optimization

Optimize CSS delivery:

```
// For Tailwind CSS, configure content purging
module.exports = {
  content: ['./pages/**/*.{js,ts,jsx,tsx}', './components/**/*.{js,ts,jsx,tsx}'],
  // Other Tailwind config
}
```

For web fonts:

```
/* Self-host fonts with subset loading */
@font-face {
  font-family: 'Brand Font';
  src: url('/fonts/brand-font-latin.woff2') format('woff2');
  unicode-range: U+0000-00FF; /* Only Latin characters */
  font-display: swap;
}

/* Consider system fonts for non-brand elements */
body {
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
}
```

Properly optimized fonts can reduce font-related bandwidth by **60-80%** while maintaining visual quality.

API Response Optimization Techniques

API responses can contribute significantly to bandwidth consumption, particularly for data-intensive applications.

Response Size Reduction Strategies

Implement field filtering to allow clients to request only needed data:

```
// api/products/[id].js

export async function GET(request) {

  const { searchParams } = new URL(request.url);

  const productId = request.nextUrl.pathname.split('/').pop();

  const fields = searchParams.get('fields')?.split(',') || null;

  const product = await getProductById(productId);

  // Return only requested fields if specified

  if (fields) {

    const filteredProduct = {};

    fields.forEach(field => {

      if (product[field] !== undefined) {

        filteredProduct[field] = product[field];

      }

    });

    return Response.json(filteredProduct);

  }

  return Response.json(product)

}
```

```
// Client usage

fetch('/api/products/123?fields=id,name,price')

  .then(res => res.json())

  .then(data => console.log(data));
```

You can also use GraphQL to reduce the response:

```
export async function GET(request) {
  // For GraphQL queries in request body
  let gqlQuery;

  try {
    // Try to read the request body as text
    const body = await request.text();
    // If body contains a GraphQL query, use it directly
    if (body && body.includes('query')) {
      gqlQuery = body;
    }
  } catch (error) {
    console.error("Error parsing request body:", error);
  }

  // If no GraphQL query was found in the body, extract product ID from path
  // and build a query based on query parameters (fallback to old approach)
  if (!gqlQuery) {
    const { searchParams } = new URL(request.url);
    const productId = request.nextUrl.pathname.split('/').pop();
    const fields = searchParams.get('fields')?.split(',') || ['id', 'name',
'price'];

    const queryFields = fields.join('\n ');
    gqlQuery = `
      query {
        product(id: "${productId}") {
          ${queryFields}
        }
      }`;
  }

  // Pass the query to your GraphQL execution function
```

```
const result = await executeGraphQLQuery(gqlQuery);

// Return the product data
return Response.json(result.data.product);
}
```

Pagination Implementation

Implement pagination to limit response size:

```
// api/products.js

export async function GET(request) {

  const { searchParams } = new URL(request.url);

  const page = parseInt(searchParams.get('page') || '1');

  const limit = parseInt(searchParams.get('limit') || '20');

  const skip = (page - 1) * limit;

  const [products, total] = await Promise.all([

    getProducts({ skip, limit }),

    getProductCount()

  ]);

  return Response.json({

    data: products,

    pagination: {

      total,

      pages: Math.ceil(total / limit),

      page,

      limit

    }

  });
}
```

```
    }  
  });  
}
```

Compression Implementation

Vercel handles most compression automatically, but you can optimize further:

```
// middleware.js  
  
import { NextResponse } from 'next/server';  
  
export function middleware(request) {  
  
  // Continue to the next middleware or route handler  
  
  const response = NextResponse.next();  
  
  // Add your middleware logic here  
  
  return response;  
}  
  
export const config = {  
  // Adding matcher eliminates the amount of middleware invocations  
  matcher: [  
    // Apply to API routes  
    '/api/:path*',  
  
  ],  
};
```

Data Normalization

Normalize and reduce the size of nested responses to eliminate redundancy:

```
// Before: Redundant nested data

const response = {

  products: [

    {

      id: 1,

      name: 'Product 1',

      category: { id: 5, name: 'Category A', description: '...' }

    },

    {

      id: 2,

      name: 'Product 2',

      category: { id: 5, name: 'Category A', description: '...' }

    }

  ]

};

// After: Normalized data

const normalizedResponse = {

  products: [

    { id: 1, name: 'Product 1', categoryId: 5 },

    { id: 2, name: 'Product 2', categoryId: 5 }

  ],

}
```

```

categories: {
  '5': { id: 5, name: 'Category A', description: '...' }
}
};

```

These API optimization techniques can reduce response sizes by 40-90%, depending on the data structure and application needs.

Prefetching Optimization Strategies

Next.js prefetches linked pages to improve the navigation experience, but this can significantly impact bandwidth usage.

Controlling Next.js Link Prefetching

Next.js prefetches links by default, which improves user experience but increases bandwidth usage:

```

// Disable prefetching for rarely accessed links
<Link href="/terms-of-service" prefetch={false}>Terms of Service</Link>

// In App Router, control prefetching behavior
<Link href="/product/123" prefetch="false">View Product</Link>

```

Prefetch configuration options:

- **false**: Disable prefetching entirely
- **true**: Default behavior, prefetch when link enters viewport

Custom Prefetching Implementation

When some pages don't get much traffic or aren't critical to the user journey, it's better to hold off on prefetching until the user shows intent, like hovering over a link. A custom logic will let you control exactly when Next.js prefetches routes:

```
function SmartLink({ href, children, ...props }) {  
  
  const router = useRouter();  
  
  const handleMouseEnter = useCallback(() => {  
  
    router.prefetch(href);  
  
  }, [router, href]);  
  
  return (  
  
    <Link  
  
      href={href}  
  
      prefetch={false}  
  
      onMouseEnter={handleMouseEnter}  
  
      {...props}  
  
    >  
  
      {children}  
  
    </Link>  
  
  );  
  
}
```

Practical Implementation Guide

To effectively reduce bandwidth costs, follow this systematic approach:

1. **Analyze current usage:** Identify your highest bandwidth consumers in Vercel Analytics

2. Optimize in order of impact:

- Start with image optimization (usually highest impact)
 - Then tackle API response optimization
 - Next, optimize JavaScript bundles
 - Finally, implement prefetch controls
3. **Measure results:** Track bandwidth reduction after each implementation to validate your approach
4. **Continue monitoring:** Regularly review analytics to catch new bandwidth hotspots as your application evolves

Case Study: Real-World Bandwidth Reduction

A mid-sized e-commerce application implemented these optimizations:

- Converted unoptimized images to WebP format
- Changed Vercel default settings for functions
- Implemented field filtering and pagination for product listing APIs
- Reducing API response size
- Shortening function execution times with DB optimizations
- Added bundle analysis, lazy loading of components, and removed unused dependencies
- Applied selective prefetching for product listings

The results:

- 62% reduction in total bandwidth usage
- 40% decrease in monthly Vercel hosting costs
- 18% improvement in page load performance

These bandwidth optimization strategies can lead to significant cost reductions for your Next.js application on Vercel. By implementing them, many teams have achieved bandwidth reductions of 30-70%, translating directly to cost savings as applications scale.

Remember that bandwidth optimization is not just about cost reduction—it also improves application performance, particularly for users on slower connections or mobile devices. The resulting improvements in page load times and overall responsiveness can lead to better user engagement, higher conversion rates, and improved search engine rankings.

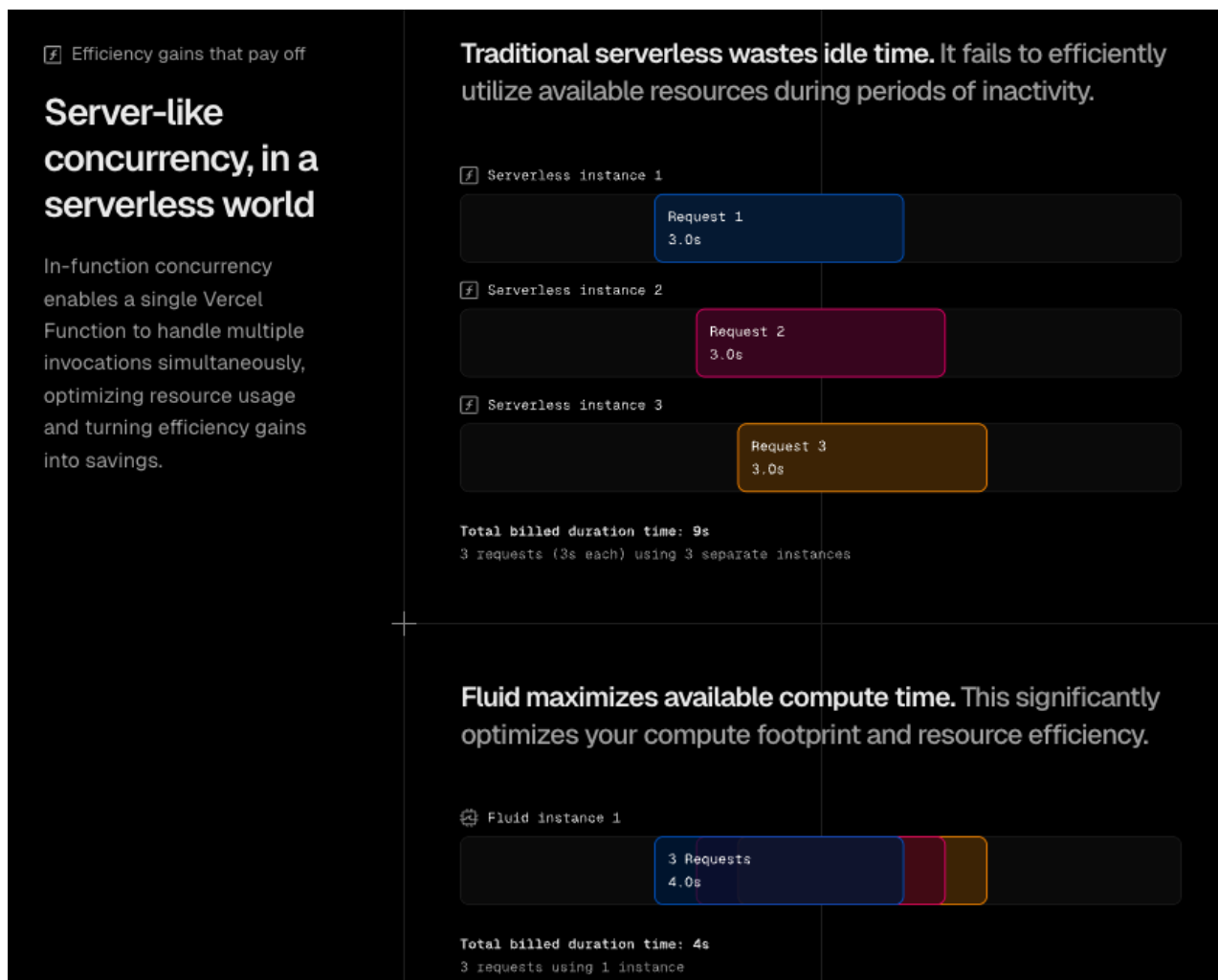
Vercel Defaults Optimization

While optimizing your code provides significant cost savings, Vercel also offers several built-in configuration options that can directly reduce your hosting expenses. These settings are often overlooked but can deliver immediate cost benefits without requiring extensive code changes.

Fluid Compute for Cost Savings

Vercel recently introduced Fluid Compute, a significant advancement in serverless function efficiency that can substantially reduce costs.

Understanding Fluid Compute Architecture



Source: Vercel

Fluid Compute intelligently reuses serverless function instances instead of creating a new instance for every invocation. In **traditional serverless**, each invocation gets a dedicated instance, while **Fluid Compute** allows multiple invocations to share instances when possible. This approach allows Vercel to significantly reduce the resource overhead per invocation, translating to direct cost savings for you.

Implementation and Configuration

Enabling Fluid Compute is straightforward:

1. Navigate to your **Vercel project settings**
2. Select "**Functions**" in the sidebar
3. Find "**Fluid Compute**" and toggle it on

When enabling Fluid Compute, there are a few considerations:

- It requires the **Standard CPU** setting (**1x**)
- Some functions with specific state management might need adaptation

Most Next.js applications can enable Fluid Compute with no code changes required. According to Vercel's documentation, this feature alone can reduce serverless function costs by **20-30%** without any other optimizations.

Monitoring Performance Impact

After enabling **Fluid Compute**, monitor your application's performance using Vercel Analytics:

- Function execution time
- Cold start frequency
- Overall application response time

Most applications see maintained or improved performance after enabling Fluid Compute, but it's important to verify this for your specific application.

New Image Optimization Pricing Model Strategies

Vercel has recently updated its image optimization pricing model from a source-based model to a transformation-based model. Understanding and optimizing for this new model can significantly reduce costs.

Understanding the Transformation-Based Pricing

Under the new model, you're billed based on unique image transformations rather than source images. A transformation is now defined by a unique combination of width, height, format, and quality. That means reducing the number of unique transformations directly reduces costs. Let's say you're using Image optimization in your project. Vercel will present you with an estimation of new costs when you try to transition.

Optimization Strategies for the New Model

To optimize for this pricing model:

1. **Standardize image dimensions:**

```
// Before: Many custom sizes
<Image src="/product.jpg" width={213} height={142} />
<Image src="/banner.jpg" width={1204} height={402} />

// After: Standardized sizes
<Image src="/product.jpg" width={200} height={150} />
<Image src="/banner.jpg" width={1200} height={400} />
```

2. **Limit quality variations:**

```
// Define consistent quality settings
const imageLoader = ({ src, width, quality = 75 }) => {
  return `${src}?w=${width}&q=${quality}`;
};
// Use consistent quality across the application
<Image loader={imageLoader} src="/image.jpg" width={800} height={600} />
```

3. **Consider external image services** for high-volume sites:

```
// Custom loader using an external service

const cloudinaryLoader = ({ src, width, quality }) => {

  return
  `https://res.cloudinary.com/your-account/image/fetch/w_${width},q_${quality} ||
  75}/${src}`;

};

// Usage

<Image loader={cloudinaryLoader} src="/image.jpg" width={800} height={600} />
```

This approach can significantly reduce your image optimization costs, especially for applications with many images displayed in standardized layouts.

Preview Deployment Optimization

Preview deployments can consume significant resources, especially for active development teams:

```
// vercel.json

{

  "github": {

    "silent": true,

    "autoJobCancelation": true

  }

}
```

Additional strategies:

- Implement automatic cleanup of old preview deployments
- Consider limiting which branches generate previews
- Use environment variables to disable data-intensive features in previews

Production-Specific Settings

In production, prioritize performance and reliability:

```
// vercel.json for production
{
  "functions": {
    "api/*.js": {
      "memory": 512
    }
  }
}
```

Consider slightly higher memory allocations in production compared to preview environments, as the cost-to-performance benefit is more important for end users than for internal testing. Tuning Vercel's default settings is one of the fastest ways to cut costs. It usually takes minimal development work and can lead to immediate savings.

Adjusting function configurations, turning on Fluid Compute, standardizing image transformations, and setting environment-specific options can reduce your Vercel bill by **20–40%**. Compared to code-level changes, these tweaks are quick to implement and don't require much testing, which makes them a great place to start.

What to Do Next

Vercel gives you a powerful platform to build, scale, and deploy Next.js apps, but that power comes at a cost. While traffic drives up bills, they can rise because of small inefficiencies, default settings, and development habits that go unnoticed until it's time to pay up.

If you've read this far, chances are you're already thinking about where to start. Maybe you've already spotted a few quick wins. But if your team doesn't have time to do a full audit, or if you want a second set of eyes on your current setup, we can help.

We offer a focused **Vercel Cost Optimization service**, designed specifically for Next.js teams spending \$1,500 or more per month. In only two weeks, Pagepro's CTO will personally review your usage, apply high-impact optimizations, and help you reduce costs without rewriting your app or sacrificing performance.

The time you invest in optimizing today could save your team thousands tomorrow, so let's make it count.



Jakub Dakowicz



Chris Lojniewski

[BOOK A MEETING](#)