

April 20, 2025

PPR for everyone with the Progressive Rendering Format



Raees Iqbal
Founder

Today, we're excited to introduce the Progressive Rendering Format, an open specification designed to bring edge-accelerated streaming and partial rendering capabilities to any web framework. This format builds on our work implementing [edge-accelerated Partial Prerendering for Next.js](#), but takes a crucial step forward by making these powerful capabilities available to all frameworks as an open standard.

Open Web vs. Proprietary Edge Solutions

Modern web applications face a fundamental challenge: balancing the delivery of an instant initial experience with the need for dynamic, personalized content. Traditional approaches have forced developers to choose between:

- Static generation (fast initial load, but limited dynamic content)
- Server-side rendering (dynamic content, but slower initial load)
- Client-side rendering (delayed content visibility and poor SEO)

Next.js recently introduced Partial Prerendering (PPR) to address this challenge, but there's a catch: Vercel's implementation relies on proprietary edge capabilities only available on their platform. This creates a situation where one of the web's most powerful new rendering patterns is locked behind a closed platform.

We believe this approach to emerging web standards is fundamentally incompatible with the open nature of the web. The web's greatest strength has always been its openness and interoperability—critical features shouldn't be limited to a single framework on a single platform.

From ESI to Progressive Rendering Format

The concept of assembling web content at the edge isn't entirely new. Edge Side Includes (ESI), a technology dating back to the early 2000s, attempted to solve similar problems by using an XML-based templating language embedded within HTML. While ESI saw some adoption, it predated modern JavaScript frameworks and wasn't designed for today's streaming-oriented web architecture.

Vercel's implementation of PPR can be seen as a proprietary, modern take on ESI principles—assembling static and dynamic content at the edge, but with closed implementation details and framework-specific constraints.

The Progressive Rendering Format builds on these foundations but takes a fundamentally different approach: it standardizes the protocol for edge-based content assembly into an open, JSON-based format that any CDN, framework, or hosting provider can implement. This creates a truly open ecosystem for one of the web's most promising new rendering patterns.

What is the Progressive Rendering Format?

At its core, the Progressive Rendering Format is a JSON-based specification that provides a standard way for edge networks and CDNs to efficiently deliver progressively streamed web pages. It consists of:

1. A response format specification that separates static and dynamic content
2. A processing mechanism that serves static content immediately from edge nodes
3. A streaming protocol for progressively loading dynamic content

Unlike ESI's XML-based approach embedded directly in HTML, or Vercel's closed implementation tied to Next.js, the Progressive Rendering Format is designed to be framework-agnostic and platform-independent—any web framework or hosting provider can implement it.

The Technical Specification

The Progressive Rendering Format uses a structured JSON format to define how content should be delivered. For a complete technical reference, see our [format specification documentation](#).

Progressive Rendering Format	JSON	
<pre>1 { 2 "v1": { 3 "status": 200, 4 "headers": { 5 "content-type": ["text/html; charset=utf-8"] 6 }, 7 "body": [8 { 9 "text": "<!DOCTYPE html><html><head>...</head><body><div id='root'><!-- Static shell c content -->" 10 }, 11 { 12 "remoteBody": { 13 "relativeUrl": "/api/dynamic-content", 14 "method": "GET", 15 "forwardRequestHeaders": true 16 } 17 }, 18 { 19 "text": "</div></body></html>" 20 } 21] 22 } 23 }</pre>		

The format includes three key components, each offering significant advantages over traditional approaches like ESI:

1. Response Metadata

The top level includes HTTP status and headers to apply to the final response, allowing fine-grained control over caching and other response characteristics. This provides much more flexibility than ESI's limited header handling capabilities.

2. Body Segments

The body array contains segments that make up the complete response, which can be either:

- **Static text:** HTML content served instantly from the edge cache
- **Remote bodies:** Dynamic content fetched from origin servers or APIs

Unlike ESI, which uses fixed fragment substitution, the Progressive Rendering Format enables true streaming delivery where static content begins flowing to the client immediately while dynamic content loads in parallel.

3. Dynamic Content Resolution

The `remoteBody` objects include powerful features for resolving dynamic content that go far beyond what's possible with ESI:

- URL resolution with request-aware variables
- Request header forwarding
- Custom headers and request method specification
- Support for various HTTP methods

This approach allows edge networks to immediately deliver the static shell from cache while asynchronously fetching and streaming the dynamic portions—creating a truly progressive rendering experience that maximizes perceived performance.

URL and Request Variables

The format includes special variables to reference properties of the original request. Our [technical documentation](#) provides a complete reference table of all available variables.

Request Variables Example

JSON



```
1 {
2   "remoteBody": {
3     "relativeUrl": "$requestPath",
4     "method": "$requestMethod",
5     "additionalHeaders": {
6       "x-original-url": "$requestUrl",
7       "x-original-host": "$requestHost"
```

```
8     }  
9     }  
10 }
```

This allows for powerful dynamic behavior while maintaining a cacheable static shell. Unlike proprietary solutions, these variables are fully documented and standardized across all implementations of the format.

Edge Processing: How It Works

The magic happens at the edge where CDN nodes:

1. Recognize Progressive Rendering responses (via the `progressive-rendering-format` header)
2. Parse the format and begin streaming static content immediately
3. Fetch dynamic content from origin servers in parallel
4. Stream the combined response progressively to the client

This results in dramatically faster perceived performance—the initial HTML shell is delivered instantly from the closest edge node, while dynamic content streams in as it becomes available. But unlike proprietary solutions, the Progressive Rendering Format can be implemented by any CDN or hosting provider, not just a single platform.

Why Open Standards Matter for the Web

The distinction between Progressive Rendering Format and proprietary alternatives has significant implications for the web ecosystem:

- **Framework Flexibility:** Not tied to a single framework like Next.js—works with any JavaScript framework
- **Platform Independence:** Can be implemented by any hosting provider, not just a single vendor
- **Community Governance:** Open for community contributions and evolution, not controlled by a single company
- **Documented Standards:** Fully documented format with transparent specifications

- **Long-term Stability:** Less risk of vendor lock-in or unexpected changes to proprietary features

By adopting the Progressive Rendering Format, developers get all the performance benefits of edge-accelerated partial rendering without sacrificing their freedom to choose frameworks and hosting providers.

Benefits for All Web Frameworks

The Progressive Rendering Format delivers significant advantages across all frameworks:

- Faster Time to First Byte (TTFB) through edge delivery of static content
- Improved Core Web Vitals scores
- Reduced origin server load
- Better scalability during traffic spikes
- Support for personalized content without performance penalties

We will be working with framework maintainers to develop adapters for the format.

Working with the Format Today

For complete technical details, refer to our [Progressive Rendering Format documentation](#). The format supports multiple implementation approaches to fit different use cases:

1. Using Next.js PPR on Tilda

The easiest way to get started is with Next.js PPR on Tilda. Our implementation already uses the Progressive Rendering Format under the hood as an open alternative to Vercel's proprietary implementation. Simply deploy your Next.js project with PPR to Tilda:

Enable PPR on Next.js

Bash



```
1 tilda build nextjs --ppr=15-canary-v1
```

2. Build-time Static Implementation

For maximum performance, you can pre-generate Progressive Rendering Format JSON files during your build process and configure routing to serve them with the appropriate headers. This approach is ideal for any framework and produces the best performance results.

Our [implementation guide](#) provides detailed examples of this approach.

3. Dynamic Server Implementation

For applications that need to generate the Progressive Rendering Format dynamically at request time, your server can:

1. Set a format-specific header (e.g., `progressive-render: 1`)
2. Add appropriate cache control headers for optimal edge caching
3. Return a JSON response following the Progressive Rendering Format

When deployed on supporting infrastructure, the edge network will automatically process these responses according to the specification, delivering static content instantly from edge nodes while streaming dynamic content in parallel.

Unlike proprietary solutions that may use vendor-specific headers, the Progressive Rendering Format works with standard HTTP headers and can be implemented with any CDN or edge platform that supports the specification.

Contributing to the Format

Progressive Rendering Format is an open specification, and we're seeking contributions from the web development community. We believe the future of web performance lies in collaboration across frameworks and platforms, not in proprietary features locked to specific vendors.

We'll be publishing a formal specification repository in the coming weeks, where you can:

- Review the full specification details
- Contribute enhancements and extensions
- Participate in discussions about future evolution
- Build framework adapters and integrations

The Future of Web Rendering

We believe the Progressive Rendering Format represents the future of web application delivery—combining the best aspects of static, server-side, and client-side rendering in a flexible, streaming approach that puts performance first.

By making this an open format rather than a proprietary feature, we hope to foster innovation across the ecosystem and improve performance for all web users, regardless of their preferred framework or platform. While proprietary platforms may continue to offer closed implementations, the Progressive Rendering Format ensures that advanced rendering capabilities remain available to everyone as part of the open web.

Getting Started

Ready to experience the benefits of open, standards-based Progressive Rendering Format?

- Read our [comprehensive technical documentation](#)
- Try [Next.js PPR on Tilda](#) today
- Review implementation examples in the [developer guide](#)
- Sign up for updates on framework adapters
- Join our Discord community to discuss the format

We're excited to see what you'll build with Progressive Rendering Format, and how together we can make the web faster for everyone—not just users of specific frameworks or platforms.

Resources

[Pricing](#)

Online

[X \(Twitter\)](#)

Company

[Blog](#)



[Pricing](#)

[Blog](#)

[Docs](#)

[Contact](#)

|

[Dashboard](#)



[Terms of Service](#)