Task 1

Shopware 6 Performance Optimization

Case Study: Axcom (Mobile)

URL: https://pagespeed.web.dev/analysis/https-www-axcom-shop-de/k5led4w1vd?form_factor=mobile

Goal : Analyze mobile performance issues and propose concrete technical solutions.

Key Performance Problems Identified

1. Slow First Contentful Paint (FCP)

- Measures how fast the first visible content appears
- Strongly affects perceived speed and bounce rate

2. Slow Largest Contentful Paint (LCP)

Main causes:

- Large hero images / media
- Slow server response (TTFB)
- Render-blocking CSS & JavaScript
- Late loading of above-the-fold elements

Case Study: Main Technical Causes

From PageSpeed analysis:

- **Render-blocking CSS & JavaScript**
- Large unused CSS & JS bundles
- Heavy plugin assets
- Tracking scripts blocking main thread.
- No **preload prioritization**

Common Optimization Solutions (Overview)

1. **Minify / Optimize Images & Combine CSS and JS files**
   Convert images to WebP format for better compression and quality, which helps reduce file sizes without losing clarity.
2. **Minify & Remove Unused CSS & JS**
   Clean up unnecessary code to improve load times and keep things running smoothly.
3. **Use Caching**
   Shopware 6 has built-in caching, but we can also:
   - Enable long-term browser caching for files
   - Use file versioning
4. **Fix Render-Blocking Resources (CSS & JS)**
5. **Lazy Loading for Images & Videos**
   Instead of loading everything at once, let images and videos appear only when needed.
6. **Use a CDN**
   A CDN like Cloudflare can serve your store's assets from the closest server, reducing load times.
7. **Disable Unnecessary Plugins**
   Some plugins slow down your store more than they help. Debug and try disabling the ones you don't need and test your speed after each change.

Focus Area: Render-Blocking Resources

Main performance bottleneck: CSS & JavaScript blocking browser rendering

Solutions

1. Fix CSS Loading Strategy

Problem:
It blocks rendering until the entire CSS file is downloaded and parsed.

To fix this, I use a two-step approach:

Step 1 – Inline Critical CSS

Extract only the CSS required for the above-the-fold content (header, hero section, first layout) and inline it directly into the HTML `<head>`.
This allows the browser to render the visible part of the page immediately, which significantly improves First Contentful Paint.

```
<style>/* critical above-the-fold CSS */</style>
```

Critical file generation using `critical`:

```
critical https://site \
  --width 375 \
  --height 812 \
  --inline false \
  --target critical-mobile.css
```

Desktop example:

```
--width 1366 --height 768
```

Mobile example (realistic iPhone X / Android):

```
--width 375 --height 812
```

Step 2 – Load the Full CSS Asynchronously

The remaining CSS is loaded using preload and applied after the page has already rendered:

```
<link rel="preload" as="style" href="theme.css"
onload="this.rel='stylesheet'">
```

---

2. Fix JavaScript Loading Strategy

Keep JS deferred (already in Shopware 6)

```
<script src="storefront.js" defer></script>
```

- Download in parallel
- Execute after HTML parsing
- No render blocking

Use async only for independent scripts

```
<script src="analytics.js" async></script>
```

This allows the browser to load and execute them whenever they are ready, without waiting for other scripts.

Delay tracking scripts

```
<script type="text/plain"
data-delay-src="/test-tracking.js"></script>

window.addEventListener('load', () => {
  setTimeout(loadTracking, 3000);
});
```

Result

- First page load without waiting for full CSS and JS
- The page renders immediately
- JavaScript and CSS do not block rendering
- Faster FCP and LCP
- SEO safe
- Better mobile user experience

Rendering Flow – CSS

Before (default CSS loading)	After (Critical CSS + Async CSS)
HTML ↓ Download full CSS (blocking) ↓ Parse CSS ↓ **Render page** ↓ Download & run JS	HTML ↓ Inline Critical CSS ↓ **Render page** (above-the-fold) ↓ Download full CSS in background ↓ Apply full styles ↓ Run JS (deferred)

Rendering Flow – JavaScript

Before (blocking JavaScript)	After (optimized JavaScript loading)
HTML parsing ↓ Download JS (blocking) ↓ Execute JS ↓ Continue rendering	HTML parsing ↓ Download JS in parallel (defer) ↓ Render page ↓ Execute JS after DOM ready ↓ Load tracking scripts after page load