Progressive Web-app (PWA) dashboard

IMT4887 - Specialisation in Web Technology

Art Jørstad Olson
artjo@stud.ntnu.no

December 2025

1 Introduction

1.1 What is the progressive web-app about?

The Artomic Røntgen machine-panel progressive web app (PWA) translates exploratory prototypes—based on research from 2020–2021 and originally shown as limited-function Figma sequences—into a coded research prototype. The earlier project explored interface designs that support task flow in radiographic imaging. In this iteration, those concepts are expressed as a minimally specified but functional machine-panel dashboard. The app's architecture is designed to prioritize energy efficiency and accessibility, and this report documents how these performance characteristics can be measured and benchmarked[5].

The dashboard's user interface includes a task-bar and several navigation elements designed for a machine-panel tablet environment. These components provide the core interaction structure for the application.

The system is implemented in two parallel environments. The initial version is a LAMP-based functional prototype built with HTML, CSS, JavaScript, and PHP. This version will support a hosted backend, with SQL data managed through phpMyAdmin. The app will also be able to collect user data.

(a) Node/Express PWA dashboard prototype with Node.js worker threads and service-worker enabled cache and image processing

The second and now primary environment is built on Node.js using the Express.js framework. It shares the same static resources and SQL database as the LAMP prototype, allowing both versions to operate concurrently during development. This Node architecture also supports service workers, push notifications, and Node worker threads, enabling efficient, non-blocking background processing.

Together, these implementations provide a flexible development setup that supports both legacy compatibility and progressive migration toward a Node-centric deployment model. As research prototypes, they are

meant to support the exploration of functionality and features and to collect data that is useful for web technology development, coding competency, and targeted user groups. These are designed to be functional research prototypes to assist in further specification, education, and to advance best practices in web technology and UX/UI development.

The app adopts a naive approach to prototyping, as specified by user and service design research [7]. The 'naive' part lies in allowing gadgets and functionality to be presented and tested in minimally-specified, tentative, or under-specified ways to gain insights about actual usability issues. This is not strictly user-centered design development; rather, it takes cues from service design while addressing progressive enhancement and responsive-design considerations from a developer standpoint.

The code prototype dashboard UI is set up as a dedicated device for viewing on tablets. It has a responsive layout and is tailored for use on three tablet configurations (to start). As such, the layout specification is created with CSS flexbox for specific adaptive view-port sizing.

1.2 The conceptual "diagnostic" imaging space

Image quality is critical for patient- or specimen focused diagnostic tasks. Screen sizes should not be too small to properly assess the details of diagnostic imaging.

The tablet PWA can serve as a controller for various technical procedures in the task flow, including monitoring the machine's temperature, adjusting ergonomic settings, selecting beam patterns, viewing calibration settings, and completing diagnostic reporting.

The central image subject represents 'a diagnostic sample' as geometric forms in a CT imaging machine gantry bay. The CT modality example presents a rotating 360-degree view of this space.The isometric gantry image set represents rotational views of the subject at 30 degree increments.

Image formats are controlled for diagnostic reporting by a variety of users: physicians for diagnosis, imaging specialists, radiography technicians, as well as other nursing support staff, caregivers, and orderlies.

Efficient machine states can be maintained in standby mode during the pre-diagnostic setup of equipment, as well as for non-critical demonstrations, instruction, technical service, or scheduling. During these tasks, the interface and diagnostic subject views are set to a properly optimized (energy-conserving) image preview.

Also, color-coded overlays and labeled image sets prevent confusion regarding attributed and correlated diagnostic notes and reports. There are typically separate inputs from radiographic imaging, nursing, aftercare, and scheduling stages of a workflow.

1.3 The Sustainable Web Manifesto

The Sustainability Manifesto proposes six guiding principles for sustainable web development; that, as designers and developers, we should aim for clean, efficient, open, honest, regenerative, and resilient solutions. There are aspects of all these principles in the project.

Initially, the focus is on specifying clean and efficient practices and building sustainable relationships with developers and designers in web technologies and healthcare.

1.3.1 Energy efficient programming languages

Compiled languages, such as JavaScript and PHP, rank among the most energy-efficient for algorithmic problems and for manipulating strings and arrays[3][9].

Clean and efficient standards and benchmarks are considered at all levels of the project, including servers, supporting data centers, and code optimization practices.

2 Measures of Impact

2.1 (LCP) Largest Contentful Paint and (FCP) First Contentful Paint

The isometric gantry image set represents rotational views of the subject at 15 or 30 degree increments. Each image in the set represents the Largest Contentful Paint (LCP) for that state of view in the dashboard UI. The First Contentful Paint (FCP) is the first image loaded by default. [3]. As such, this image is specifically scrutinized for loading performance benchmarks.

The performance scores are quite good for .png and .svg.

1.2 s for a (109 Kb) .png compared to 237 ms for a (28.8 Kb) .svg; with extreme optimization, .svgs can achieve loading efficiencies of 698 ms on (10.1 Kb). Cycling through 12 or 24 image sets compounds the resource load. Interestingly, the ultra-low file size does not necessarily load faster for individual images, but the total set of images can be optimized for edge processing on smart devices.

A performance of 2.5 seconds or less for the largest contentful paint (LCP) is considered optimal[13], but targets are set for load-times as low as possible at each quality marker.

Further, Node.js I/O is already non-blocking, and image loading can utilize its event loop to handle I/O asynchronously.

fs.readFile and Promise.all facilitate asynchronous and concurrent loading.

Three sets of images

2.1.1 Data transfer and loading performance for accessibility page

After navigating registration and login, the transfer of the accessibility data page for first-time visitors is recorded at:

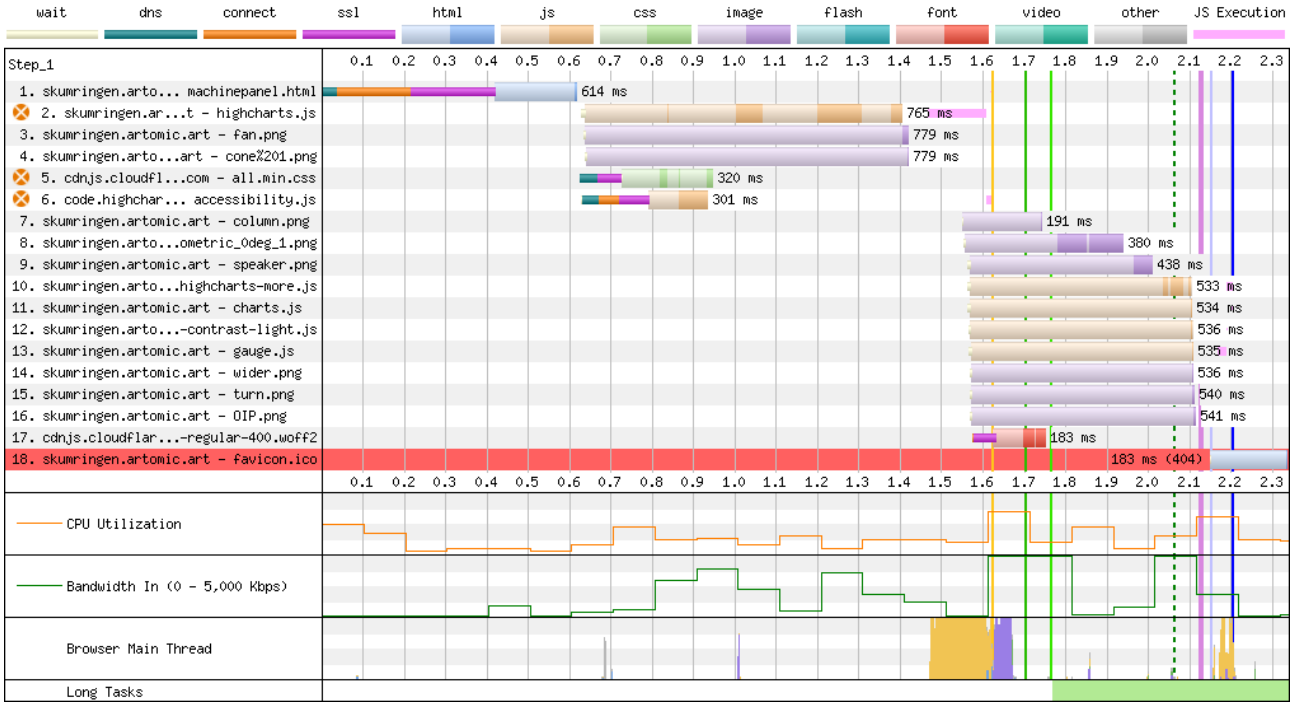
203 kB / 254 kB transferred , 834 kB / 946 kB Finish: 1.46 s, DOMContentLoaded: 1.24 s , Load: 1.47 s

Preparations included server-loaded modules for the charts and accessibility modules. Images are optimized for task priority and efficiency.

2.1.2 Data transfer and loading for dashboard

The web app dashboard is designed for a dedicated tablet screen and is responsive to designated screen sizes.

738 kB / 922 kB transferred , 834 kB / 1,119 kB resources Finish: 231 ms , DOMContentLoaded: 179 ms , Load: 232 ms



(a) page load waterfall for PHP/SQL dashboard 10/19/2025, 11:14:12 PM [14]

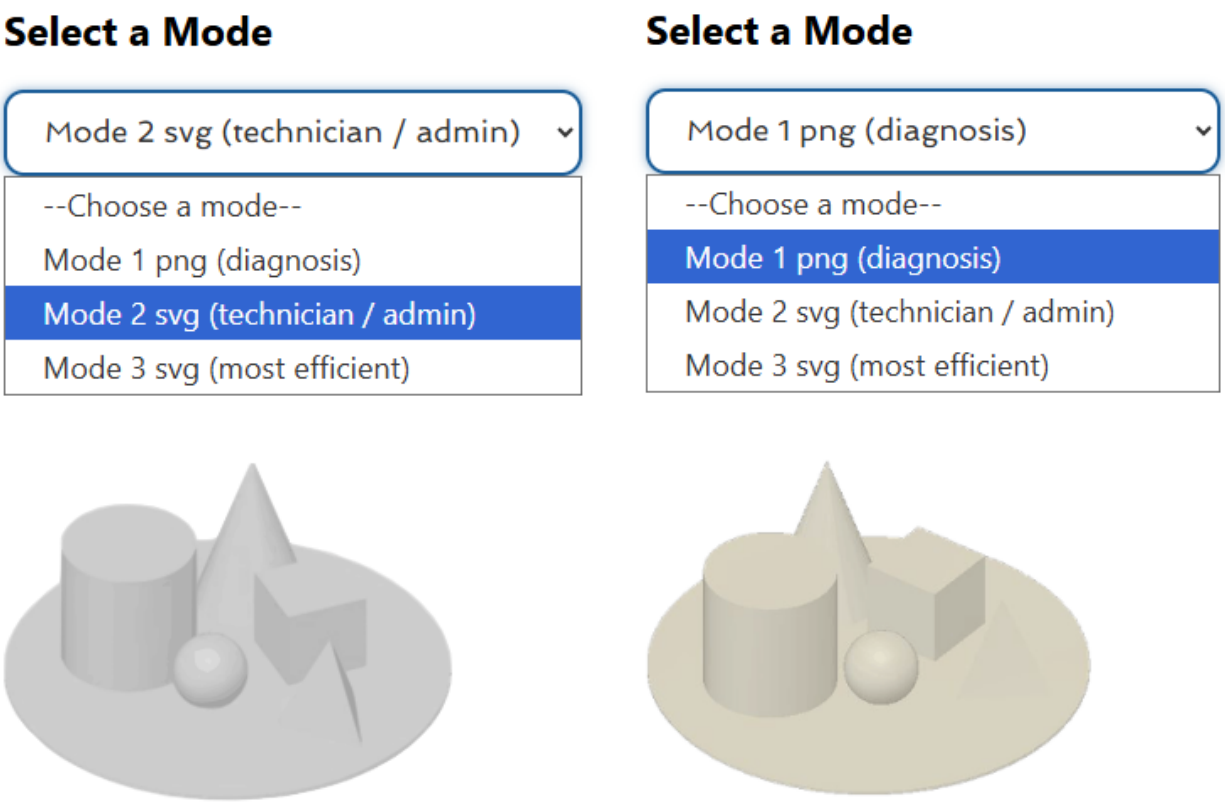
Early waterfall analysis of page loading posts great performance metrics, as well as some breakpoints indicating scripts that delay the loading of other resources. The external JavaScript libraries were downloaded and are served locally as static assets to improve reliability and load performance. There is also a font-awesome link that is best served locally. Additionally, JavaScript scripts from Highcharts that provide dashboard indicators and accessibility charts are identified as blocking processes in the waterfall report. There is also a missing favicon image (red).

Most of the code for gages and dial scripts was "tree-shaken" for local serving, but some of the web-sourced scripts persist, resulting in a difference of approximately 200 milliseconds in the load time.

Highlights from WebPageTest indicate a page weight of 415 Kb on the first load, a Largest Contentful Paint (LCP) of 2.06 s, a First Contentful Paint (FCP) of 1.7 s, a total blocking time of 0.0 s, a cumulative layout shift (CLS) of 0.0, and a Time to First Byte (TTFB) of 0.0 [14]. Overall, the performance is quite good, with some actionable areas for improvement!

2.2 Task optimized image sets

Image formats can be selected to support different task resource requirements.



(a) SVG image

(b) PNG image

Figure 3: Mode selection for image sets

Mode 1 Image quality and fidelity for diagnostic notes are preserved in PNG. Samples in this format are rendered without lighting effects. In this sample set, images are approximately 106-108 Kb.

Mode 2 Optimized .svg files include compression (SVGOMG), and the image quality is suitable for machine setting tasks, gantry positioning, or instructional procedural illustrations.

Mode 3 Efficiency optimized .svg files are at the limits of image quality, suitable for limited devices. Images are compressed to around 20 kb each, and the full batch of subject images (12 items) is 261 kb. Demonstrates suitability for smart devices and resource-constrained devices [13].

The selector specifies the default selection for loading admin access (Mode 2). This architecture supports sustainability by loading image intensive tasks in a prioritized way. JavaScript loads images based on the task mode selection. The selections could also be preset for specific access codes. For example, diagnostic notes can be keyed to image sets. Physicians' notes are specifically coded to the reference imagery. Technicians' notes are keyed to a color cast. Compression preserves only essential image metadata. The selector script makes use of Node's inbuilt I/O asynchronous (async) management of images.

A link to the dropdown sequence and other current web-app features can be found here: Artomic registration and login.

2.2.1 Code example

The following code for the isometric image sets processes arrays of captured images at 30 (or 15) degree increments. Full sets are indexed from TDC (top dead center). For the code example, each view mode array has only 4 images per set. The prototype will receive between 12 and 24 full 360-degree views of the gantry subject.

Improvements to the code and prototype image sets will include shorter naming conventions and JavaScript functionality that retains the current indexed view when changing image modes. It currently resets to the first image.

Listing 1: Example JavaScript

```
1
2  // Trigger default selection after the page loads
3  window.addEventListener("DOMContentLoaded", () => {
4    selector.dispatchEvent(new Event("change"));
5  });
6
7  // Define image lists for each mode
8  const images = {
9    mode1: [
10     "images/PNG/isometric_set_0008_1215_isometric_90deg_1.png",
11     "images/PNG/isometric_set_0009_1210_isometric_60deg_1.png",
12     "images/PNG/isometric_set_0010_1205_isometric_30deg_1.png",
13     "images/PNG/isometric_set_0011_TDC_1200_isometric_0deg_1.png",
14   ],
15   mode2: [
16     "images/SVG/optim2/isometric_set_0008_1215_isometric_90deg_1optim2.svg",
17     "images/SVG/optim2/isometric_set_0009_1210_isometric_60deg_1optim2.svg",
18     "images/SVG/optim2/isometric_set_0010_1205_isometric_30deg_1optim2.svg",
19     "images/SVG/optim2/isometric_set_0011_TDC_1200_isometric_0deg_1optim2.svg",
20   ],
21   mode3: [
22     "images/SVG/optim3/isometric_set_0000_1255_isometric_330deg_1optim3.svg",
23     "images/SVG/optim3/isometric_set_0001_1250_isometric_300deg_1optim3.svg",
24     "images/SVG/optim3/isometric_set_0002_1245_isometric_270deg_1optim3.svg",
25     "images/SVG/optim3/isometric_set_0003_1240_isometric_ 250deg_1optim3.svg",
26   ],
27 };
28
29 const selector = document.getElementById("modeSelector");
30 const displayImage = document.getElementById("displayImage");
31 const nextBtn = document.getElementById("nextBtn");
32
33 let currentMode = "";
34 let currentIndex = 0;
35
36 // When dropdown changes
37 selector.addEventListener("change", function () {
38   currentMode = this.value;
39   currentIndex = 0;
40
41   if (currentMode && images[currentMode].length > 0) {
42     displayImage.src = images[currentMode][currentIndex];
43     displayImage.style.display = "block";
44     nextBtn.disabled = false;
45   } else {
46     displayImage.style.display = "none";
47     nextBtn.disabled = true;
48   }
49 });
50
51 // Cycle through images
52 nextBtn.addEventListener("click", function () {
53   if (currentMode) {
54     currentIndex = (currentIndex + 1) % images[currentMode].length;
55     displayImage.src = images[currentMode][currentIndex];
56   }
57 });
```

2.3 Sustainability benchmarks

2.3.1 Hosting

According to the host website, current project' servers are hosted in data centers that are 100% powered by renewable energy. As an example, their EU data center in Amsterdam runs entirely on electricity from renewable sources. Furthermore, the data center operator has signed The Climate Pledge™ and committed to reaching Net Zero emissions by 2040 [6].

IP Geolocation traces the hosting server to Phoenix, Arizona, 85001, where there are 780 hosted domains on that server.

Pinging 192.64.118.48 with 32 bytes of data: Reply from 192.64.118.48: bytes=32 time=186 ms TTL=50 ; Reply from 192.64.118.48: bytes=32 time=170 ms TTL=50 Request timed out. Reply from 192.64.118.48: bytes=32 time=166 ms TTL=50

Ping statistics for 192.64.118.48: Packets: Sent = 4, Received = 3, Lost = 1 (25Approximate round trip times in milli-seconds: Minimum = 166 ms, Maximum = 186 ms, Average = 174 ms

Listing 2:]Tracing route to node.vts.artomic.art [192.64.118.48]

```
Tracing route to artomic.art [192.64.118.48] over a maximum of 30 hops:

1      1 ms     <1 ms     <1 ms  10.0.0.1
2      5 ms      1 ms     1 ms  192.168.2.1
3      6 ms      4 ms     4 ms  36.79-160-116.customer.lyse.net [79.160.116.36]
4      3 ms      3 ms     3 ms  229.213-167-114.customer.lyse.net [213.167.114.229]
5      4 ms      *         4 ms  wiki1.lyse.net [81.167.36.40]
6     30 ms      4 ms      5 ms  219.213-167-114.customer.lyse.net [213.167.114.219]
```

The ISP gateway → local network lyse.net is a Norwegian ISP located in Stavanger.

```
7      7 ms      6 ms      5 ms  oso-b8-link.ip.twelve99.net [62.115.54.185]
8      5 ms      6 ms      7 ms  oso-b1-link.ip.twelve99.net [62.115.138.132]
9     17 ms     17 ms     17 ms  hbg-bb3-link.ip.twelve99.net [62.115.112.84]
10    25 ms     23 ms     23 ms  ffm-bb1-link.ip.twelve99.net [62.115.123.76]
11    48 ms     25 ms     26 ms  ffm-b14-link.ip.twelve99.net [62.115.132.209]
12    24 ms     25 ms     23 ms  imperva-ic-377658.ip.twelve99-cust.net [213.248.92.147]
```

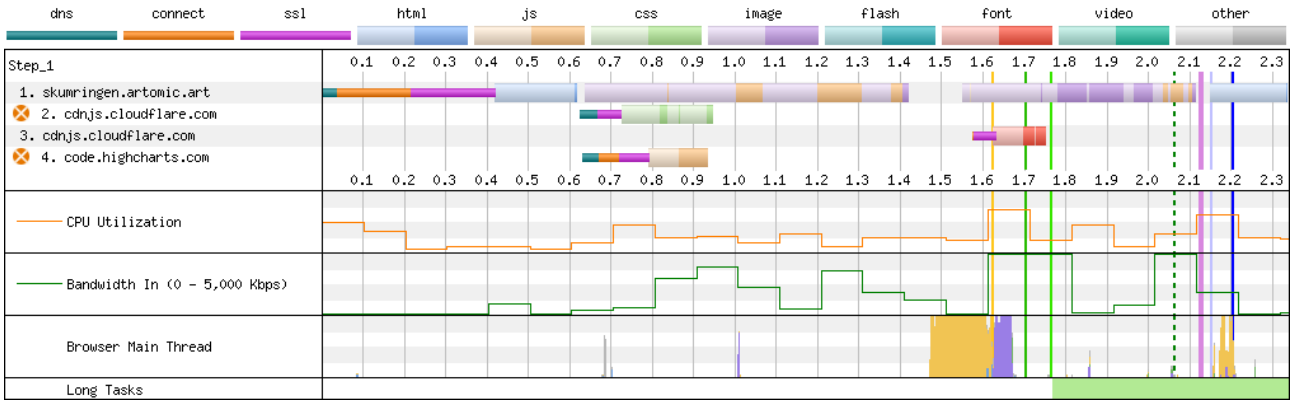
Hops 7-10 show traffic on the Telenor/Telia backbone (AS1299) from Norway to Germany. Network traffic crosses the Atlantic from Frankfurt, Germany (ffm in hop 10/11).

```
7      7 ms      6 ms      5 ms  oso-b8-link.ip.twelve99.net [62.115.54.185]
8      5 ms      6 ms      7 ms  oso-b1-link.ip.twelve99.net [62.115.138.132]
9     17 ms     17 ms     17 ms  hbg-bb3-link.ip.twelve99.net [62.115.112.84]
10    25 ms     23 ms     23 ms  ffm-bb1-link.ip.twelve99.net [62.115.123.76]
11    48 ms     25 ms     26 ms  ffm-b14-link.ip.twelve99.net [62.115.132.209]
12    24 ms     25 ms     23 ms  imperva-ic-377658.ip.twelve99-cust.net [213.248.92.147]
```

The shared hosting infrastructure of the network is responsible for regular timeouts and multiple domains on the same IP.

Latency from Norway 166–181 ms after the transatlantic hop is in-line with typical contributions of base latency plus processing overhead: Fiber path (Europe-US), Routing and peering, Firewall/ CDN, and Load balancer or host response delays. Still, the physical origin latency is a consideration for the compounded impact over time. Fewer network DNS, TCP, and TLS handshakes are efficiency goals that enhance the end-user experience regarding page loading times and the loading of first contentful paint (FCP) images. Local cache strategies, use of a CDN.

(18,000km)/(200,000km/s)/90ms (one-way)180ms RTT



(a) Connection waterfall for artomic.art dashboard 10/19/2025, 11:14:12 PM. The dns, connection, and ssl amount to at least o.42 seconds before page load. [14]

2.4 Accessibility enabled report overlays, charts, guages

Preparations include testing locally and server-loaded modules for accessibility-enabled charts and guages. Visual charts and data presentations that support screen readers are implemented via modules through the terminal (npm) and several other formats. Each requires careful consideration of its impact on runtime and loading. There are advantages to using bundlers and tree-shaking techniques when utilizing many library modules.

2.4.1 Optimizing Fonts

For special typefaces, WOFF fonts and TTF fonts are loaded onto the server, preloaded onto the page, and widely used fonts are specified as alternatives for offline display or during service interruptions.

Optimization by converting TTF fonts to WOFF2 format and uploading only the character weights used can further optimize efficiency, as can creating subsets of the font sets that strip out unused characters.

(The creation of font subsets, along with code compression, should be done closer to publication to avoid limiting future routine copy-editing and updates by preserving A-Z, 1-9, and the most commonly used special characters while dropping special characters [3].)

(a) PWA icon 11/17/25

2.4.2 Page and image caching

Setting image cache and font cache policy statements in the .htaccess file reinforces energy efficiency, prioritizes access, meets task requirements, and may serve as a preventative safety measure in the case of diagnostic images. Physicians need to work from only the relevant, current imagery when making an accurate prognosis. A no-cache policy on diagnostic sets prevents the inadvertent loading of outdated scans.

In the code prototype, 3 image classifications can be accessed with different cache policies.

```
# ----- PNG: No Cache ----- hi res diagnostic images are never cached
<FilesMatch "\.png$">
    Header set Cache-Control "no-store, no-cache, must-revalidate"
    Header set Pragma "no-cache"
    Header set Expires "0"
</FilesMatch>

# ----- Cache other images for 2 weeks -----
<FilesMatch "\.(jpg|jpeg|gif|svg)$">
    Header set Cache-Control "public, max-age=1209600, immutable"
</FilesMatch>

# ----- Cache fonts in the directory for 1 year -----
<Directory "/full/path/to/subdomain/fonts">
    <FilesMatch "\.(woff2|woff|ttf|otf)$">
        Header set Cache-Control "public, max-age=31536000, immutable"
    </FilesMatch>
</Directory>

# ----- Fonts: Cache for 1 year -----
<FilesMatch "\.(woff2|woff|ttf|otf)$">
    Header set Cache-Control "public, max-age=31536000, immutable"
</FilesMatch>

# ----- Cache other images for 1 month -----
<FilesMatch "\.(webp)$">
    Header set Cache-Control "public, max-age=2592000, immutable"
</FilesMatch>

# Proxy /benchmark
RewriteCond %{REQUEST_URI} ^/benchmark$
RewriteRule ^(.*)$ http://127.0.0.1:3000/benchmark [P,L]

# Proxy /benchmark-status
RewriteCond %{REQUEST_URI} ^/benchmark-status$
RewriteRule ^(.*)$ http://127.0.0.1:3000/benchmark-status [P,L]

# Required for proxy to work
<IfModule mod_proxy.c>
    ProxyRequests Off
    ProxyPreserveHost On
</IfModule>
```

Listing 1: Example `.htaccess` configuration - Apache .htaccess / httpd.conf configuration syntax 18/11/25

3 Plan for implementation

The plans for this project are to develop a high-quality, code-prototype web app that demonstrates sustainable design while facilitating UI research and data collection. The dashboard shell is simultaneously deployed as a Node/Express project and on a PHP/SQL/Apache stack.

Specific goals for this course are to adhere to best practices in web development regarding efficiency, performance, accessibility, and interoperability.

As naive UI/UX research dashboards, these are designed as 'a trigger-action sandbox' for protected user testing and data collection in select focus groups, for presentation, and with developers. They are meant to be

exploratory and, at this stage, minimally specified; they should never be scaled beyond the responsiveness of their principals or principles. [8].

3.1 Node implementation

Deployment testing is also performed concurrently in a Node.js (JSON) environment, utilizing the Express.js framework. There are several benefits to working with Node.js

Node-apps are supported on Namecheap; however, Node.js is apparently not loaded globally in the hosting terminal window. It is necessary to update dependencies in the package.JSON file to include 'sharp' and 'eslint'.

JavaScript: Modules: Node.js provides the path module to safely create paths, regardless of the operating system:

Web Workers:

4 Responsible JavaScript

4.1 Priority of Constituencies

As van Kesteren notes,

> "In case of conflict, consider users over authors over implementers over specifiers over theoretical purity" [12].

User-centered design is inherently iterative and assumes a perpetual, self-sustaining development cycle focused on users. In this sense, it could be considered an ideology. However, as I specify early research prototypes from user insights, the adoption of user research is tempered by pragmatic measures to ensure proper data collection and responsible handling of insights. Service design methodology emphasizes client relationships and co-design inputs, recognizing that stakeholders cannot be sidelined. Consequently, the priority of constituencies should be considered cyclical rather than strictly hierarchical, with developers actively nurturing relationships and securing commitments.

By combining modules, web-components, and trigger–action logic, client-side frameworks can sustain a clean, maintainable architecture without compromising runtime performance. As noted, "a best practice that preserves the developer experience at the expense of the user experience is not a best practice, but rather a self-serving one" (p. 31). Planning for user experience involves optimizing load times, minimizing blocking behavior, reducing memory overhead, and accommodating device limitations. Effective optimization begins by minimizing delays, page shifts, and other constraints to make them recede or disappear.

The specifications for the Machine Panel and reporting views of the dashboard are grounded in task analysis, interviews, and user research on workflow in radiographic clinics. While the project remains under-specified in some areas, it is research-driven rather than dictated solely by developer or designer imagination.

4.2 Technology statement

A project technology statement is templated as a .git supplement, specifying the technologies used, the principles guiding the project, and the conscious choices about what to avoid[1]. The Technology Statement is available at GitHub Repository.

Research Fidelity is where we balance the quality deployment of prototypes with usability testing.

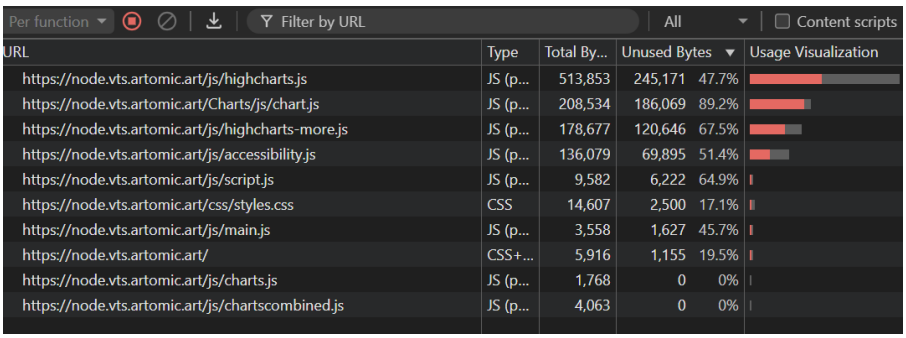
Accessibility is meant to support assistive readers and universal design, but not to inadvertently provide an 'open-source' for pre-release market-research. Information security and data-handling routines should, at all times, be followed mindfully to protect clients, contributors, and all participants.

Vanilla code supports the possibilities for development teams and progressive professional development.

4.3 Optimizing the JavaScript heap

In consideration of optimizing the JavaScript payload, I have reduced some of the libraries and consolidated the code into a compressed version.

Bloated code libraries and unused code result in slower loading times for first paint (FP) and first contentful paint (FCP) service-worker caching and offline loading times increase.



(a) The "Coverage" panel in Chrome DevTools shows inefficiency of loading full JS libraries

4.4 Built for speed

While the performance analysis, returning results of 0.015 and 0.0 blocking, suggests that scripts are not a problem, most scripts are loading asynchronously and loading in the 500 ms to 700 ms range.

In the performance waterfall, we see two batches of activity, corresponding to before and after ssl. Typically, SSL/TLS (HTTPS) is established before any page content is delivered. An exception is any inline script and preloaded or cached image that persists in the browser and is running or loaded when the browser parses the HTML.

4.4.1 HTTPS/2

- HTTP/2 is the first major HTTP protocol update since 1997, when HTTP/1.1 was first published by the IETF.It improves efficiency, speed, and security.
- HTTP/2 is binary rather than textual. It is fully multiplexed, sending multiple requests in parallel over a single TCP connection.
- HPACK header compression reduces overhead.
- Servers should “push” responses proactively into client caches instead of waiting for a new request for each resource.
- The ALPN extension allows for faster, encrypted connections since the application protocol is determined during the initial connection.
- HTTP/2 reduces additional round trip times (RTT), making loading faster.
- Domain sharding and asset concatenation are no longer needed with HTTP/2.

The ’cone’ and ’column’ images are preloaded from the cache, as well as a script for gages and dials, and an accessibility chart script. Correcting this indeed reduced page load time by up to 200 ms. These button icons are also compressed PNGs and are provided in a more efficient image format.

The highcharts.js scripts are from a library but are loaded from the server, and the code for gages has been parsed into a combined script and purged of unused code. The performance waterfall indicates that it is still blocking some other processes. In this case, the staggered blocks of processing actually perform better than all images loading asynchronously.

4.4.2 Resource profiles

```
// Serving the PHP site's resource folders as static assets
app.use('/images',express.static('/home/artomic/skumringen.artomic.art/images'));
app.use('/css', express.static('/home/artomic/skumringen.artomic.art/css'));
app.use('/js', express.static('/home/artomic/skumringen.artomic.art/js'));
app.use('/fonts', express.static('/home/artomic/skumringen.artomic.art/fonts'));
```

4.5 Progressively enhanced JavaScript

Byte for byte processing of JavaScript requires more computational power than CSS or HTML[13].

4.6 Navigating Toolchains

4.7 Github

I've had a Github account for years, but I have not yet fully utilized it. Perhaps I underestimated the functionality.

4.7.1 Task runners

For the Node project, the package.JSON file is supplemented with devDependencies 'nodemon' and 'eslint', and to the dependencies, I added 'sharp' for image scaling.

4.7.2 nodemon

Nodemon is a utility for development that automatically restarts the Node.js runtime when changes are made to the project files.

4.7.3 ESlint

ESlint is a developer specific module in Node.js that analyzes JavaScript/TypeScript code and reports and fixes problems in the code editor— as syntax errors, undefined variables, or style issues.

4.7.4 Transpilers

Transpilers like Babel transform modern JavaScript features and syntax into a legacy syntax that runs reliably in all browsers (ES5 : commonJS-style). This includes newer classes, async/await, operators such as spread syntax, arrow functions, and template literals[**?**]. A quick run of Babel reduced the size of the main JS file by about 30 lines. I'm careful here to keep the idea of'vanilla code,' untransformed (ES5) legacy syntax for JavaScript..

4.7.5 Differential serving

The module/nomodule pattern may be useful for the different versions of the dashboard in future implementations. The PHP/SQL(LAMP) version without modules, commonJS-style, and the Node.js/Express.js/SQL version utilizing modules and ES5.

4.7.6 CleanCSS and PurgeCSS

Improved loading performance with minification and code compression. I bundled 4 CSS files using CleanCSS, which outputs a formatted bundled .dev.css file OR a minified bundle. With this, PurgeCSS.can scan HTML for classes, IDs, and selector tags, purging unused CSS. It's best to maintain a .dev.css file and run a minified version prior to publishing.

4.7.7 Rollup JavaScript bundler

Rollup is a recommended bundler that publishes using ES modules. I have set up a .config file that outputs a Readable development bundle (not minified; it keeps formatting), a minified bundle, and an obfuscated bundle. I have previously used another bundler along with eslint and am currently testing this one for tree-shaking to purge unused JavaScript code in the chart library modules. The bundled and purged JS file should be much less bloated with unused code.

There seem to be some errors because these separate scripts contain parts that need to be run in a specific contingent order. Scripts previously "linted" from modules to CommonJS may have lost some functionality when recombined with ES modules for charts due to load order and dependency errors

4.7.8 Gzip Compression

Gzip compression is also implemented on the server side for the transfer of text assets. It is enabled in an .htaccess file, as well as in PHP, and optimizes page loading by compressing HTML, CSS, JavaScript, SVG, and JSON during the transfer from the server to the browser.

4.7.9 Obfuscation

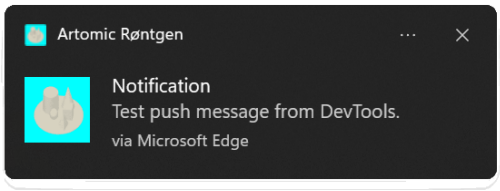
While probably not a necessary functionality, obfuscation makes code harder to read and protects against easy copying through the browser developer window.

Sensitive data submitted via forms is protected in the PHP / SQL backend. Form data is likewise protected in an Express/Node server setup by separating the code into public and private modules. The code for DOM elements, HTML, CSS, and JavaScript would likely be of particular interest to developers, as JavaScript is usually quite specific in its functionality without requiring a lot of editing.

I have tested obfuscation in a production workflow to reinforce developer routines that support freelancing and protect the site from wholesale copying.

4.7.10 Push API

The Push browser API allows the PWA to receive push notifications even in an offline state. It uses service-workers in the background and a Web Push Protocol with VAPID id. A subscription object sends a push message to the server, and the service worker receives it.



(a) Push Notification

4.7.11 webPush

A Node.js library for sending Web-Push Notifications to browsers. Works with the Push API and VAPID id.

4.7.12 VAPID

An identification system for push notifications. VAPID generates a pair of KEY: a PUBLIC key shared with the client frontend and a PRIVATE key token for signing requests. Required by browsers for security.

4.7.13 uuid

Generates universally unique identifiers (UUIDs) with version control.

4.7.14 sharp

An image processing library for resizing and converting formats (png/jpg/webp/avif) is a candidate for a web-worker or Node worker-thread process to handle background image processing tasks asynchronously.

4.7.15 .dotenv

Used to load .env environment variables into process.env to keep configurations, passwords, and access information out of shared production code.

4.8 Error handling and fault tracing

Essential to my "toolchain" are some tools and approaches for fault-tracing and error handling. I have found that the error logs produced by cPanel are useful for tracing issues, and I consult ChatGPT for various methods to test the network through the server Terminal or Node.js Terminal (REPL) windows.

4.9 Some current problems and troubleshooting

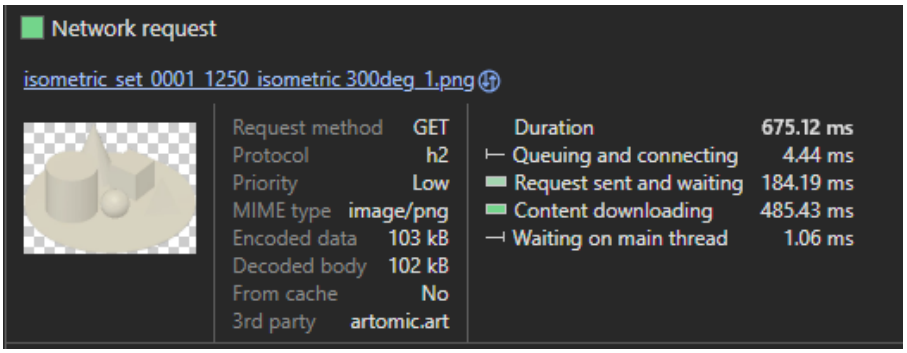
- There is a problem with the offline registration of the service-worker.
- There is a problem compiling and tree-shaking highcharts.js from the contingent modules.

4.10 cPanel Terminal and Node(PWA) Application Manager

Plenty of package updates, configuration, and error handling are done through the cPanel terminal, and the npm restart of the PWA is performed in a Node Application Manager window. Proper versioning and installations are done here in the command line interface.

4.11 Smoother runtime performance

Runtime performance describes how responsive a page is to user interactions. The loading time of the .PNG diagnostic images is notable for benchmarks. The image loading time ranges from 365 to 675 ms for each rotational view. Meanwhile, the .svg optimized files barely register when loaded onto the network.



(a) Runtime loading benchmark on .png (diagnosis) image [14]

4.12 Thermal throttling

Combating thermal throttling on resource limited devices involves optimizing loading throughout the runtime. Mobile devices have passive cooling; therefore, the processing of large image sets should be split up, and images should be queued in a smart way.

As of October 2025, StatCounter cites that 34.4% of devices worldwide are Android, while 14.7% are iOS. iOS devices are typically high performance devices, while Android runs the full gamut of performance. Resource limited devices are constrained by thermal throttling [13][11].

When we can bridge the performance gap between high end and low end devices, we support accessibility and extend our range of services.

4.13 Description of Use of Artificial Intelligence

Over the past few years, I have gradually integrated AI tools into my workflow—both as a developer and to support academic writing. AI has become an indispensable resource for debugging, tracking, clarifying code, and troubleshooting errors. In general, I try to use AI in a granular way: to build, reinforce, and test my own understanding of prompts. I often ask ChatGPT "sideways" questions to expand on what I already know and to deepen my comprehension rather than replace it.

The current web development project is built with vanilla code, and I make a deliberate effort to strengthen my competence by understanding the underlying data structures and syntax. My Technology Statement favors vanilla code over libraries and frameworks that obscure core developer competencies. There is always a risk in allowing AI to become a cognitive crutch; therefore, the litmus test is to build responsibly—ensuring that I can manage and understand every layer of the code I create.

In academic writing, I have traditionally drawn a clear distinction between conventional copy-editing and authorship. Increasingly, however, AI is used to suggest phrasing and refine language. Even so, all content is reviewed, edited, and academically assessed by me—line by line. AI has not been used to generate entire paragraphs or complete assignments. Instead, I use it to reformulate and clarify my initial ideas, ensuring that the final writing reflects my own authorship and critical engagement.

5 The Artomic Rontgen Progressive Web App (PWA)

The benefits of a progressive-web app go beyond "appy-ness".

6 Accessibility enabled charts

Accessibility enabled charts from MIT and Highcharts.com show data in screen-reader-accessible balloons that expand on hovering over areas of the charts[4]. The chart from www.chartjs.org is from an open-source library[2].

6.1 Responsive sizing

iPad mini 768 x 1024 iPad Pro 11" 834 x 1194 Desktop PC 1440 x 1024

6.2 Performance Budgeting

I have approached this project from the start with an eye toward performance and efficiency. Performance budgeting is designed to track the network AND device. The aim is to create an extremely clean, efficient, and fast-loading app prototype (App-shell).

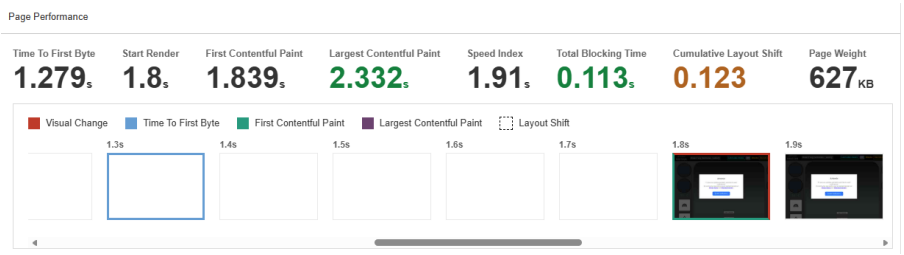
In general, a target performance goal is set as follows: 5 seconds or less for the first-load Time-to-Interactive (TTI) and less than 2 seconds for subsequent loads. We constrain ourselves to a real-world baseline device and network configuration to measure progress. The observed global baseline is a $200 Android device on a 400 Kbps link with a 400 ms round-trip time ("RTT"); a 'ping'. This translates into a budget of approximately 130-170 KB of critical-path resources, depending on the composition [10].

Execution of JavaScript that controls interactions, processes inputs, and button actions, as well as the construction of the DOM and the building of the layout, must take place on the main thread. Other tasks, including the parsing of critical path resources such as images, HTML, CSS, JavaScript, and other calculations, can be delegated to background or worker threads [10].

A budget should be set at 170 KB for critical path resources.

TBT is considered good if kept under 300 ms. [13].

After regular loading, the user can initiate benchmark testing of tasks to demonstrate non-blocking server-side task splitting and processing using Node.js worker threads. On the prototype, the tasks are started and delegated to 4 worker threads, which do not block other interactions, such as rotating or changing sample-subject image modes.



(a) Performance benchmark 11/17/25
[14]

6.3 Accessibility and Responsive layout

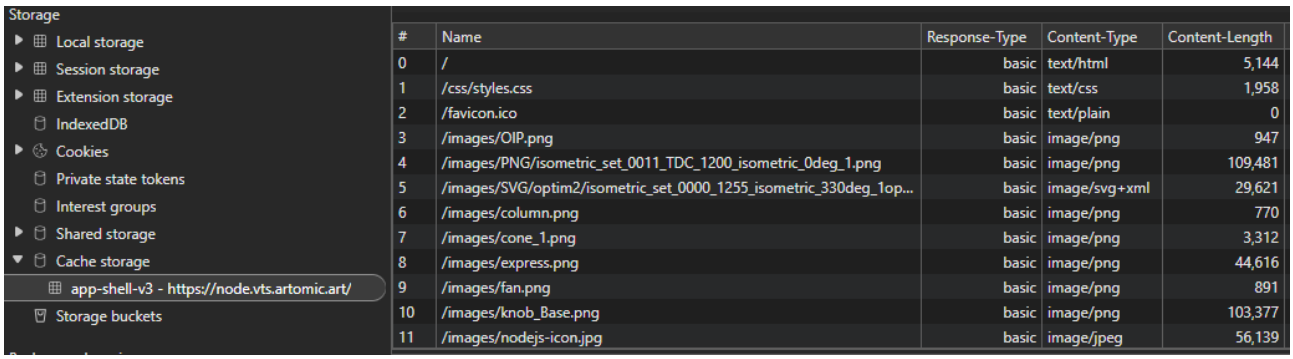
CSS Flexbox and media queries provide responsive display sizing for several tablet sizes. While the app is intended for tablet viewing, we may extend the adaptive formatting for demonstration on mobile devices.

6.4 Node.js worker threads

The Node.js runtime uses worker-threads to delegate CPU intensive processing to multiple threads. The PWA demonstrates background processes and displays a progress bar showing how 4 workers process a long task without blocking the main event-loop.

6.5 A service worker

A service-worker handles push notifications and app-shell caching. It responds to events in the browser, and in this implementation, it caches a versioned dashboard app-shell. After the initial page loading, the cache strategy results in a nearly instantaneous page load.



#	Name	Response-Type	Content-Type	Content-Length
0	/	basic	text/html	5,144
1	/css/styles.css	basic	text/css	1,958
2	/favicon.ico	basic	text/plain	0
3	/images/OIP.png	basic	image/png	947
4	/images/PNG/isometric_set_0011_TDC_1200_isometric_0deg_1.png	basic	image/png	109,481
5	/images/SVG/optim2/isometric_set_0000_1255_isometric_330deg_1op...	basic	image/svg+xml	29,621
6	/images/column.png	basic	image/png	770
7	/images/cone_1.png	basic	image/png	3,312
8	/images/express.png	basic	image/png	44,616
9	/images/fan.png	basic	image/png	891
10	/images/knob_Base.png	basic	image/png	103,377
11	/images/nodejs-icon.jpg	basic	image/jpeg	56,139

(a) Service worker enables caching of the PWA app-shell. Assets can be viewed in DevTools => Application

6.5.1 Fluid images

Caching of the app-shell is made possible with service-workers and provides faster page loads on subsequent requests. Some other strategies to explore with image caching or versioning include importing, loading, parsing, and sizing new versioned 'diagnostic-subject' image sets. These can be handled as non-blocking processes.

Scenario	DOMContentLoaded	Load Event
First load, 3G, no cache	16,644 ms	19,268 ms
Second load, 3G, cached	4,970 ms	7,003 ms
First load, no throttling, no cache	2,058 ms	2,316 ms
Second load, no throttling, cached	1,194 ms	1,408 ms

Table 1: Page load performance under different network and caching conditions 11/17/25

6.6 First Load, No Throttling:

The page load tests were conducted on an unthrottled Ethernet/Wi-Fi connection, with measured speeds of 106 Mbps for download and 82.6 Mbps for upload (tested via Speedtest.net).

On a fast, uncached network, the page achieves DOMContentLoaded in approximately 2.1 seconds and full load in approximately 2.3 seconds. This represents the baseline performance for a first-time visitor with no cached assets. Most resources—including scripts, styles, and the chart library—are fetched from the network, parsed, and rendered efficiently, with only a small difference (200 ms) between DOMContentLoaded and the full load, indicating that additional resources (images, fonts, or other deferred assets) are lightweight.

6.7 Second Load, No Throttling (Cached Assets):

With browser caching and the service worker in place, the second load is significantly faster, reaching DOMContentLoaded in approximately 1.2 seconds and a full load in approximately 1.4 seconds. The majority of critical resources are served from the local cache, reducing network fetch time and CPU parsing overhead. This demonstrates the clear benefit of caching: returning visitors experience a much snappier page load, even under optimal network conditions, with nearly a 40% reduction in total load time compared to the first load.

6.8 Simulated 3G performance testing

Testing performance on Google's simulation includes a 400 ms RTT and 400-600 Kbps of throughput (plus latency variability and simulated packet loss).

Under 3G throttling with a hard reload and no cached assets, the browser reports a DOMContentLoaded time of approximately 16.6 seconds and a full load time of approximately 19.3 seconds. These numbers are expectedly high because every resource—including scripts, styles, and the service worker—must be fetched over a severely limited connection with no caching benefits. The results do not reflect normal user performance but rather demonstrate the worst-case conditions for first-time visitors on a slow network.

With cached assets and the service worker in place, the page loads dramatically faster: DOMContentLoaded drops to approximately 5 seconds, and the full load completes in approximately 7 seconds under the same 3G throttling. This reflects the benefits of locally cached scripts, styles, and other static resources, meaning subsequent visits require far less network transfer. These numbers represent a much more realistic experience for returning users on slow connections, showing that the service worker and caching strategy are functioning as intended.

```
const nav = performance.getEntriesByType('navigation')[0] || performance.timing;
const start = nav.startTime ?? performance.timing.navigationStart;
const domMs = (nav.domContentLoadedEventEnd ??
    performance.timing.domContentLoadedEventEnd) - start;
const loadMs = (nav.loadEventEnd ?? performance.timing.loadEventEnd) - start;

console.log(`DOMContentLoaded: ${Math.round(domMs)} ms`);
console.log(`Load: ${Math.round(loadMs)} ms`);
```

6.9 Web manifest

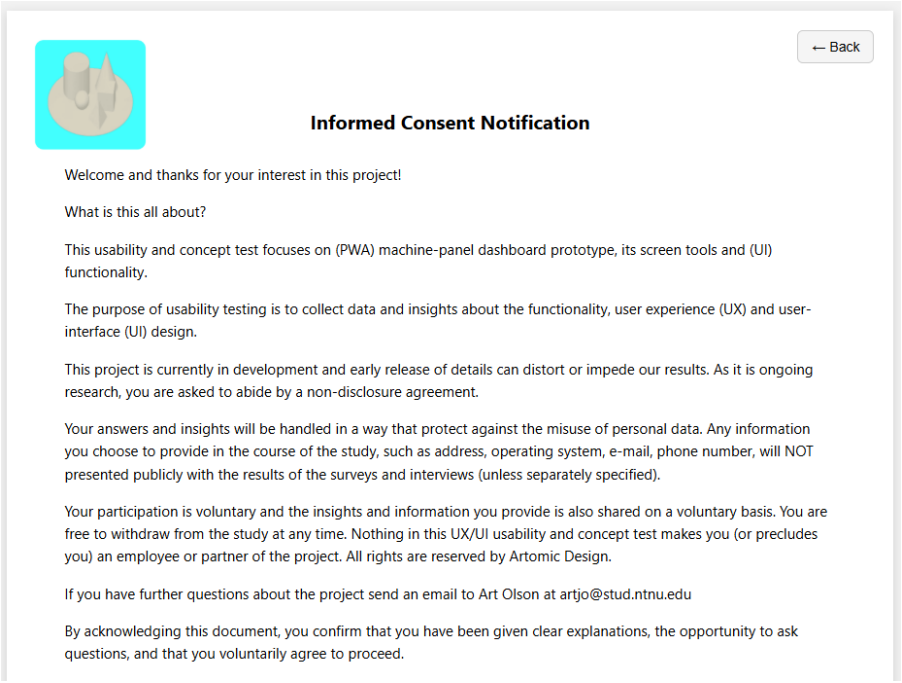
Multiple sized buttons and the "web manifest" file for the Progressive Web App (PWA) declare "no-chrome" full tablet immersive view (display-mode: full-screen) or (display:standalone).

The dash functions as a Single Page Application (SPA).

6.9.1 privacy statement and informed consent

The PWA now includes a Privacy Policy and Informed Consent notification, presented as a modal, which also functions as a release for voluntary participation in prototype user-testing and experimental data collection.

Push notifications are currently set to validate the installation of the app.



(a) Data privacy and Informed Consent notification on page load 11/17/25

6.9.2 More sustainability features, measures and methods to incorporate

- image pre-load batch sizing using 'sharp' and task division using service workers
- promises for dynamic importing of scripts & handling asynchronous operations
- PWT Authentication and user roles. The Node.js architecture
- improved offline functionality (with service-workers)
- An enabled form completion button implementation can provide enhanced functionality with a graceful (partial fallback) — the core HTML form works on all browsers.

References

[1] Artomic2020. Technology statement and project overview. https://github.com/Artomic2020/Rontgen-web-app-machine-panel/blob/main/README.md, 2025. Accessed October 2025.

[2] Chart.js Contributors. Chart.js v4.5.1, 2025.

[3] Tom Greenwood. *Sustainable Web Design*. A Book Apart, Berkeley, CA, 2021.

[4] Highsoft AS. Highcharts javascript charting library. https://www.highcharts.com, 2025. Licensed under the Highcharts License.

[5] Johanna Johansen. Imt4887 – specialisation in web technologies. University Course, Autumn 2025, 2025. Course guidelines for the semester project. Email: johanna.johansen@ntnu.no.

[6] Namecheap, Inc. What is a green internet? how tech is shaping a sustainable future. https://www.namecheap.com/blog/whats-a-green-internet/, 2021. Accessed: 2025-09-22.

[7] Art Jørstad Olson. Visualizing research prototypes for a service dashboard and machine panel in radiographic workflow. Master's thesis, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, 2021. Institute of Design.

[8] Artomic Design Art J. Olson. Artomic registration and login. https://skumringen.artomic.art/; https://www.node.vts.artomic.art/, 2025. Accessed: 2025-09-19.

[9] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE'17)*, pages 256–267, Vancouver, BC, Canada, 2017. ACM.

[10] Alex Russell. Can you afford it?: Real-world web performance budgets. https://infrequently.org/2017/10/can-you-afford-it-real-world-web-performance-budgets/, October 2017. Accessed: 2025-11-17.

[11] Statcounter Global Stats. Operating system market share worldwide, 2025.

[12] Anne van Kesteren, Maciej Stachowiak, and HTML Working Group. Html design principles. W3C Working Draft WD-html-design-principles-20071126, W3C, nov 2007. Section 3.2: "Priority of Constituencies". Accessed 16 October 2025.

[13] Jeremy Wagner. *Responsible JavaScript*. A Book Apart, Berkeley, CA, 2021.

[14] WebPageTest. Performance analysis chart. https://www.webpagetest.org/, 2025. Accessed October 2025. Chart generated using WebPageTest online performance testing tool.

Appendix

Here are some additional materials that support the main text.

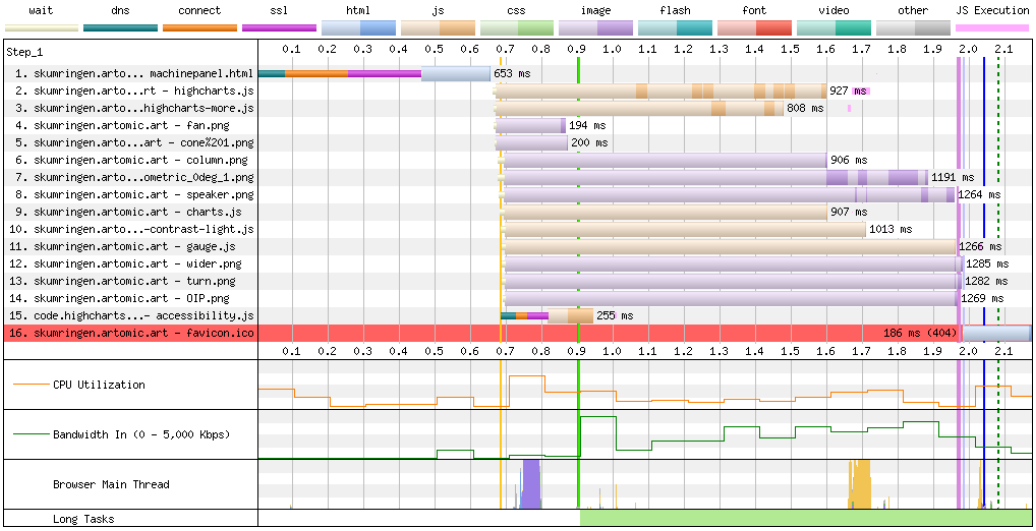


Figure 12: Modified page load waterfall 20.10.25

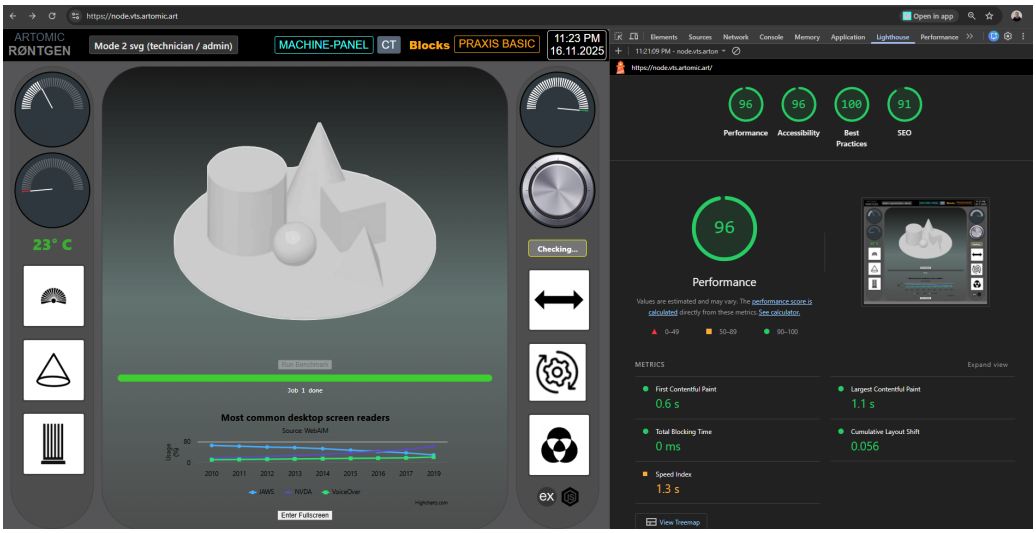


Figure 13: Performance 20.10.25

.0.3 Notes on HTTPS

This is a description of how HTTPS is actually handled for a Node/Express PWA on cPanel: Apache does SSL + static files; Node runs locally behind it:

With Apache and Node as a reverse proxy, ensure that Apache serves /service-worker.js and does not pass /service-worker.js to Node unless Node is serving it correctly. Apache (or LiteSpeed) handles HTTPS/SSL. Apache listens on port 443, serving the PWA over HTTPS using the certificate installed via cPanel.This ensures that all visitors receive a secure connection without needing Node to handle SSL. The Node app runs on HTTP internally. Node just listens on a local port (PORT 3000).No SSL configuration is required inside Node. Apache acts as a reverse proxy. Requests to certain paths (such as /benchmark) are forwarded to Node. Apache handles encryption, while Node handles the application logic.