Dynamic Caching Micro-frontends with Service Workers

Mesfin Tegegne

2025 Laurea

Laurea University of Applied Sciences

Dynamic Caching Micro-frontends with Service Workers

Mesfin Tegegne
Business Information Technology,
Developing Digital Services
Thesis
June, 2025

Laurea University of Applied Sciences **Abstract**
Business Information Technology,
Developing Digital Service
Bachelor of Business Information Technology

Mesfin Tegegne
Dynamic Caching Micro Frontend with Service Workers
Year 2025 Number of pages 54

This thesis studies how service workers can work with micro-frontend architectures to improve web application performance through dynamic caching. The main goal is to help developers and small business owners, like Mon'adi Ravintola, create fast and reliable websites. It focuses on practical solutions for better user experience and offline functionality.

The study includes a case study where Mon'adi's website was redesigned using React and TypeScript in a micro-frontend structure. Service workers were added to manage caching, which made the website load faster and work offline. Tools like Webpack 5 and performance metrics, such as Time to Interactive and First Contentful Paint, were used to measure improvements.

The theoretical section examines modern web development techniques, highlighting micro-frontend architectural concepts and the foundational elements of Progressive Web Applications, with particular attention to service workers and their associated caching mechanisms.

Findings indicate that integrating service workers within a micro-frontend architecture significantly enhances web application performance, achieving a 52% faster page load times and full offline support. The study concludes that adopting service workers in micro-frontend architectures offers substantial benefits, including improved performance and scalability. However, challenges such as implementation complexity and coordination between frontend modules are acknowledged.

Keywords: Micro-frontends, Service Workers, Dynamic Caching, Progressive Web Applications, Web Performance Optimization

Contents

1 Introduction

In the rapidly growing digital world, frontend applications are playing important role in delivering fast and responsive user experience. The demand for scalable, performant, and resilient applications has led to the adoption of innovative architectural patterns. Among these, micro-frontends have emerged as a compelling approach, enabling them breaking down a large, single frontend application into smaller, manageable parts. Each of these parts can be developed and deployed independently, making it easier to manage and update the application over time. This study specifically investigates how we can integrate micro frontend apps with dynamic caching techniques, utilizing service workers, that enhance the performance, responsiveness, and offline functionality of web applications.

To meet these objectives, a practical case study is conducted by redesigning a website for Mon'adi, a local Italian restaurant. This project demonstrates how these techniques can effectively benefit business owners by providing seamless user experiences to their websites regardless of connectivity. To enhance the clarity and readability of this thesis, AI tools such as ChatGPT and Grok were used to assist with refining the text. These tools provided support in structuring content and improving language, while all technical content and analysis remain the author’s original work.

Ultimately, this thesis serves as a reference for future development efforts, particularly for those interested in implementing micro frontend architectures that leverage service workers for caching and offline capabilities.

The structure of the thesis is organized as follows: Chapter 1 explores the background and scope of this thesis, Chapter 2 presents the theoretical foundation, exploring micro-frontends and modern web architecture while chapter 3 covers caching strategies, service workers and their implementations with respect to performance Improvement. Chapter 4 focuses on how to measure performance metrics in modern web browsers and other tools, Chapter 5 details Mon’adi’s implementation, Chapter 6 provides test results of the website with service workers as a performance testing tool also compares without service workers. Chapter 7 summarizes findings and recommends for future work.

1.1 Background and Motivation

Since the introduction of the World Wide Web by (Berners-Lee 1989) the way websites are developed and used has changed a lot. Initially, web pages were static and offered limited user interaction, but over time, they evolved into sophisticated, interactive applications designed to meet growing user expectations (Agarwal & Sastry 2022, 71-72).

Modern web development practices have shifted towards achieving modularity, scalability and enhanced user experience. However, these changes have also introduced new challenges, such as increased complexity, performance issues, speed, and latency (Paresh Kapuriya 2022). As applications become larger and involve more dependencies, maintaining performance and responsiveness becomes more difficult.

To address these issues, the concept of micro-frontends (Altexsoft 2022) emerged, allowing large web applications to be broken down into smaller and independently developed components, leading to better scalability and maintainability (Florian Rappl & Lothar Schöttner 2024, 19-20). In parallel, technologies like service workers have been introduced to improve web performance, particularly for offline functionality and efficient caching. Service workers act as a proxy between the browser and the network (Google Chrome 2024c) however, they are different from traditional proxy servers. Service workers are client-side script, helping applications load quickly even under poor network conditions (MDN Web Docs, 2024b).

This thesis came from the need to help small businesses like Mon'adi Ravintola. Using micro-frontends and service workers can make their websites fast and reliable. The case study shows how these ideas work in a real project.

1.2 Objectives and Scope

This thesis wants to show the combination of micro-frontends and service workers to enhance website performance through effective caching strategies. The case study is not in a step-by-step micro frontend application development tutorial type, rather its focus is on exploring the combination of micro-frontends with service worker for caching purpose.

The scope covers caching and performance, not backend microservices or full coding tutorials. It uses React, TypeScript, and Webpack 5 for the website. Performance is checked with metrics like Time to Interactive and First Contentful Paint.

Building a micro frontend application is a complex process that involves considerable effort, from setting up individual micro apps to deploying them, but this thesis centers specifically on caching aspects of the micro frontend project.

2 Foundations of Micro-frontends and Web Architecture

This chapter is exploring foundational knowledge of micro-frontends and web architecture in general, focusing on both the frontend and backend components and how they relate to modern web development practices.

2.1 Web Application Architecture

The frontend, or client side, of a website includes user-facing elements that enable interaction (Ritesh et al., 2023). It aims to provide an engaging experience through visual components like text, images, and buttons. Krause (2024, xxv) notes that frontends are built using HTML for structure, CSS for styling, and JavaScript for dynamic user-browser interactions. For the Mon'adi Ravintola Website, HTML organized the content, CSS improved its look, and JavaScript added interactivity like menu functions and performance metrics.

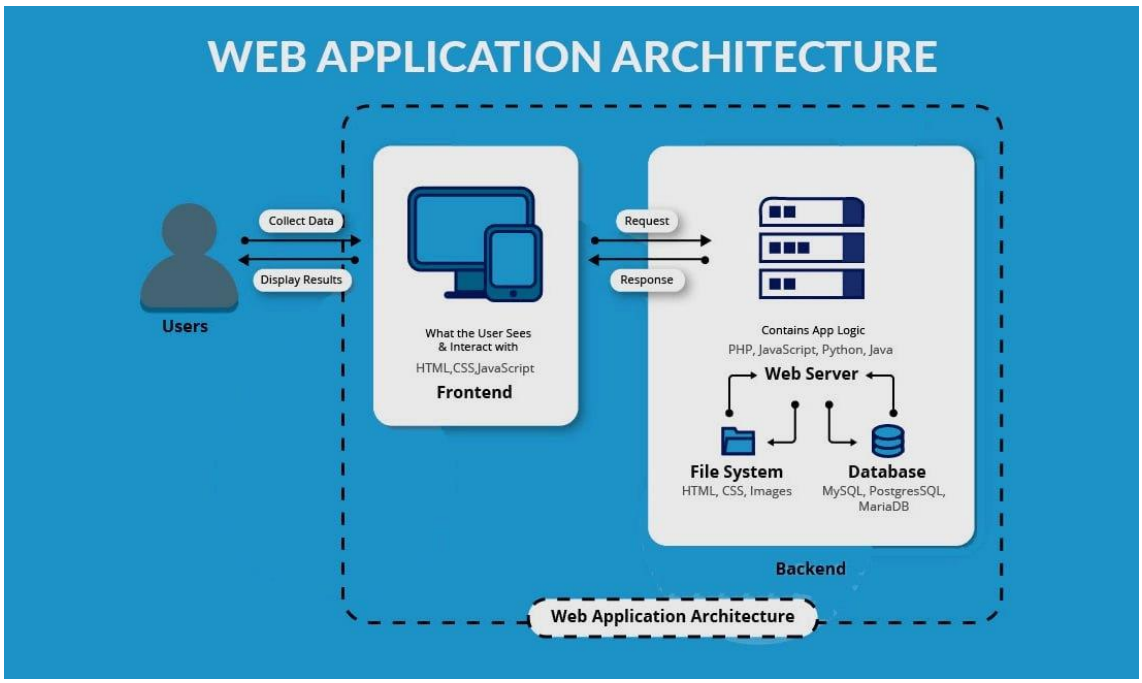


Figure 1: Modern Web Application Architecture. (adopted from Balcany 2024).

But for a full-fledged website, the frontend is only one side of the application; to have dynamic behavior, backend such as database and server implementations is needed. According to Martin Krause (2024), the Backend represents the behind-the-scenes of an application, providing the core logic that is not directly accessible by users. Interaction occurs only through specific frontend components that communicate with the backend. Codecademy (2024) popular online coding bootcamp states backend applications can perform operations such as Read, Create, Update, and Delete; these are commonly referred to as CRUD operations. As Figure 1 depicts A user can communicate with server requesting for data,

which is a read data operation, Since The server in the backend has access to data storages or file systems to fetch data requested by user and responded back to the user in user friendly text usually with HTML format. Backend programs are typically written in programming languages such as JavaScript, Typescript, PHP, Python, and Java etc.

While the frontend and backend build websites, micro-frontends split the frontend into smaller parts for better development that represents an architectural style where a user Interface (UI) is divided into smaller portions, each portion is independently developed and deployed. At the end, these portions are ultimately integrated to form a complete application. This division of the development process remains hidden to end users, who will still access a complete, seamless web application.

In traditional web development trend, building a complex web page from the ground up as a monolithic application can be inefficient, especially due to tightly coupled components (Micro Frontend 2022). These challenges have led developers to adopt micro-frontends architecture.

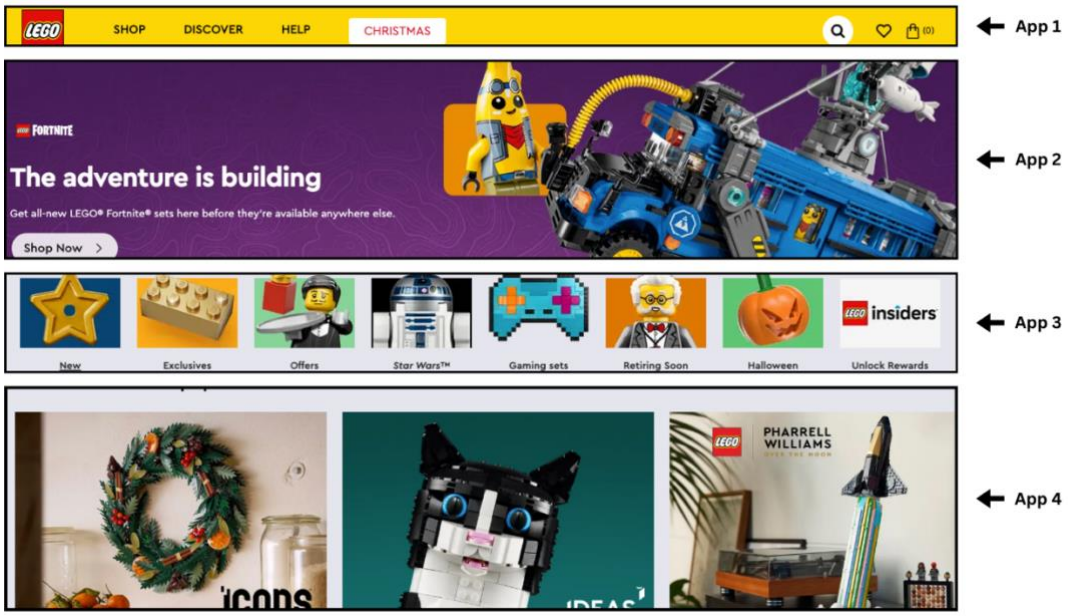


Figure 2: A Lego web page represented as a micro frontend app. (adopted from LEGO 2024).

Figure 2 illustrates the LEGO website as an example of a micro frontend-based web application, divided into four distinct parts. Each app can be developed by different teams using different programming languages. One benefit of this approach is fault isolation, if one of the apps fails, the rest of the web page remains functional and accessible.

Unlike backend microservices, micro-frontends focus on the user interface, but they work together through APIs (Vinci J. Rufus 2023, 4). Additionally, microservices have been around longer and have more established practices compared to micro-frontends. As shown below in

Figure 3, microservices and micro-frontends are communicating through RestAPIs or GraphQL over the internet to access any needed data from the backend services for presentation in the frontend. Since microservices are outside the scope of this thesis, they will not be discussed in further detail. This thesis looks at frontend splitting. Mon'adi's website uses these ideas to create a fast and interactive experience

2.2 Monolithic vs. Micro-frontends Architecture

Monolithic architecture (Craig, P., n.d.) is a development approach where the entire website is built as a single, unified entity. It integrates various elements, such as the user interface, application logic, and data management, into a tightly coupled system. A key limitation of this method is that modifications to one part often affect other parts, potentially causing widespread complications across the application. For the Mon'adi Ravintola Website, this approach was initially considered but later replaced with a more modular structure to avoid such dependencies and enhance development flexibility.

As illustrated in Figure1, a typical monolithic architecture, that if the server responsible for responding to user requests fails, it will cause the entire application to become unresponsive. As a result, error messages being displayed to the users. This is unfavorable to user experience, as it fails to meet the user expectations seamless functionality.

However, the monolithic approach also has its advantages, particularly for smaller web applications where simplicity and rapid development are prioritized over scalability. It is also easier to manage due to its lower complexity and the entire codebase being in one location, making it simple to maintain and work with (Atlassian 2024).

In contrast, micro-frontends divide a web application into smaller, independently developed and deployed components, each component are responsible for a different feature or section of the UI. The following table summarizes the key differences between monolithic architecture and micro-frontends for a clearer understanding of their benefits and drawbacks.

Table 1: Comparison between Monolithic and Micro frontend architecture. (adopted from Vivek Shukla 2023).

Feature	**Monolithic Architecture**	**Micro-frontends**
Codebase	Single unified codebase	Multiple modular components
Coupling	Tightly coupled	Loosely coupled

Fault Isolation	Errors can impact the entire application	Faults are contained within individual components
Scalability	Difficult to scale as project grows	Scalable through independent components
Development Speed	Faster initial development	Slower setup, but faster parallel development later
Team Collaboration	Requires tight coordination	Teams work independently with greater autonomy
Complexity	Lower for smaller projects	Higher due to distributed architecture

The key differences between monolithic architecture and micro-frontends are evident in several aspects.

In a monolithic architecture, a failure in one development layer can cause the entire webpage to crash, whereas micro-frontends confine errors to the specific application in which they occur, preventing them from affecting the rest of the system (Vinci J. Rufus, 2023, 21). Additionally, scaling monolithic applications becomes challenging as the codebase grows, but micro-frontends facilitate scalable development by allowing each component to be scaled independently based on its specific requirements (Micro Frontend, 2022). Furthermore, in a monolithic architecture, teams must coordinate closely since all features reside within the same codebase, which restricts autonomy and can result in longer deployment cycles. In contrast, micro-frontends empower teams to work independently, deploy components separately, and utilize different technologies when necessary, leading to greater autonomy and faster deployment cycles (Vinci J. Rufus, 2023).

While monolithic websites have limitations, micro-frontends offer a modular solution, especially for complex projects. However, Micro-frontends are not solution for every type of development, it’s essential to evaluate the complexity, size of developers and maintainability aspects before deciding whether to adopt micro-frontends approach for development.

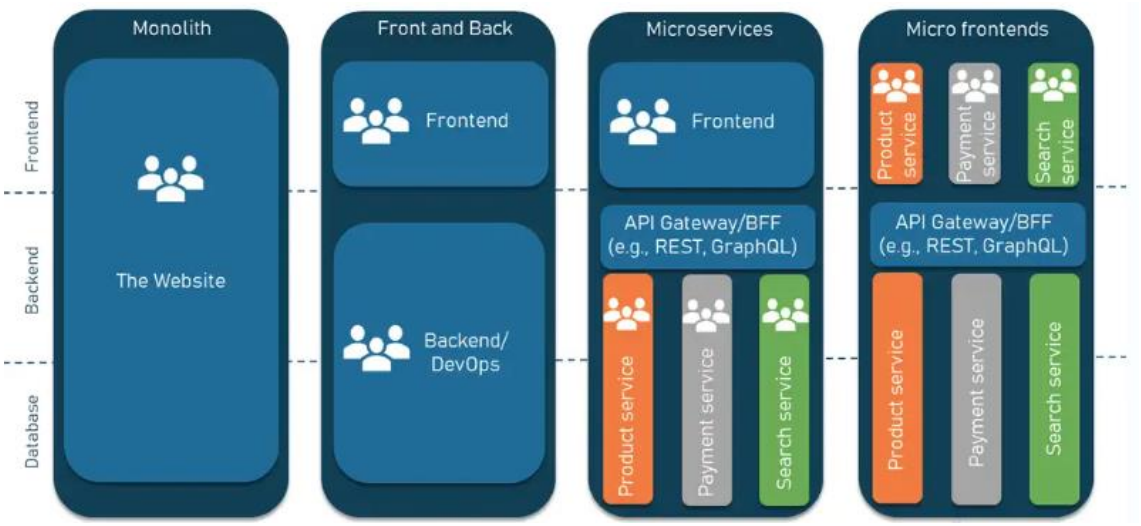


Figure 3: Web development architectural approach. (adopted from micro-frontends 2022).

As Figure 3 Illustrates a hypothetical e-commerce project that showcases the transition from a traditional monolithic web application to a microservices architecture, ultimately progressing to micro-frontends. Initially, the application adopts a monolithic structure, with the frontend, backend, and database closely intertwined as a single unit. Although this method is straightforward for smaller applications, it becomes challenging to manage as the project scales due to its tightly coupled design (Navdeep Singh Gill, 2024).

To adopt micro-frontends, Mon'adi followed a step-by-step transition from its original monolithic design. In general, as the above figure 3 shows, the transformation journey is undertaken with three stages as explained below.

Separation of Concerns: From monolith to Front-and-Back Separation. The first step in breaking down a monolithic application is to separate the frontend from the backend. This separation makes both parts more independent, enabling each to be developed and deployed individually. As a result , development teams can iterate faster without creating botlenecks due to tightly coupled components (Craig Parravicini n.d.).

Backend Decomposition into Microservices: The backend is then broken down into microservices, while the frontend stays as a single unit. The backend is split into smaller, specialized services like Product Service, Payment Service, and Search Service, each handling a distinct function of the application. This process enhances scalability and modularity, allowing each service to be managed independently (Micro Frontend, 2022).

Frontend Decomposition into Micro-frontends: In the final stage, the frontend is divided into micro-frontends, where individual frontend components, such as Product App, Payment App, and Search App, are developed and deployed separately. Each micro frontend corresponds to

an associated backend service. For example, the Product App in the frontend communicates directly with the Product Service in the backend. This allows different teams to work on specific features independently, ensuring fault isolation and enabling faster, more flexible updates (Vinci J. Rufus 2023, 21).

2.3 Integration and Orchestration

Integration in the micro-frontends context refers to connecting each individual app to ensure that they work together seamlessly. In Figure3, we assume an eCommerce web application is composed of three different micro-frontends and corresponding microservices. Once the applications are ready from each distinct team, the next step is to ensure they integrate seamlessly and work as a unified system.

The three types of integration methods for micro-frontends each present distinct strengths and drawbacks, with no single method being universally ideal; the choice depends on project requirements and personal preferences. Build-time integration necessitates rebuilding and redeploying the root application whenever changes are made to any application, which can result in repetitive work and extended deployment cycles. This approach may not suit projects requiring high modularity. However, it offers the advantage of tight version control, which is critical when multiple application components are interdependent (Vinci J. Rufus, 2023, 20-21).

In contrast, run-time integration enables independent deployment of micro-frontends without strong dependencies, fostering greater scalability and decoupled workflows. In this method, micro-frontends are composed at runtime, providing enhanced flexibility but requiring careful orchestration. This approach was adopted in this thesis to achieve improved scalability and maintainability, aligning effectively with the principles of modular development (Vinci J. Rufus, 2023, 21-22).

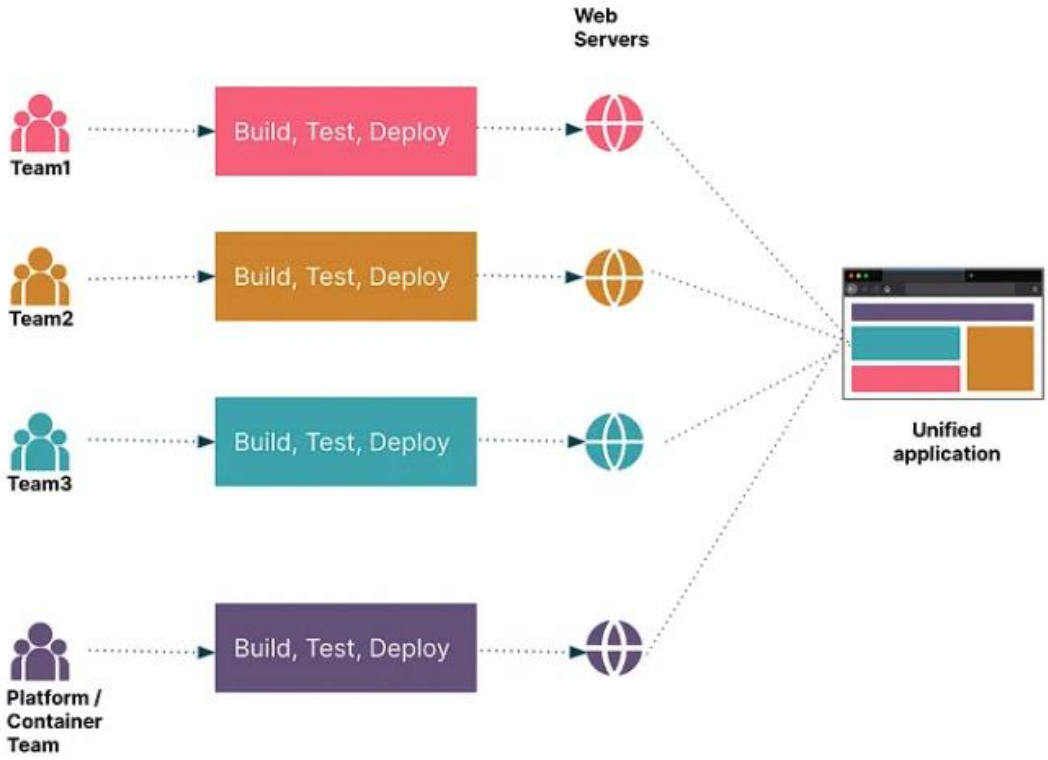


Figure 4: Run-Time Integration. (adopted from Vishal Sharma 2023).

As illustrated in Figure 4, each micro frontend application is designed and deployed by distinct teams, often utilizing different programming languages during the development phase (Chameera Dulanga, 2024). To facilitate this process, tools such as Webpack Module Federation, Turborepo, and Single SPA are commonly employed to integrate these applications into a single cohesive unit (Vishal Sharma, 2023). Additionally, server-time integration involves various server-side tasks to achieve effective integration. This method supports server-side rendering, which can lead to faster perceived load times for users. However, it introduces added complexity on the server side, particularly when managing multiple micro frontend modules (Udemy, 2021).

To achieve run-time integration, tools like Webpack 5's Module Federation make it easy to share code between apps. Each individual micro frontend application is orchestrated to ensure a smooth deployment and scalable application process.

Webpack 5 serves as a leading tool for bundling JavaScript applications, proving particularly effective for projects such as micro-frontends. Its module federation feature enables the sharing of code across independently built applications, enhancing modularity and flexibility, which makes it a preferred choice for large-scale, modular projects (Webpack, 2021). As depicted in Figure 5, Webpack bundles dependencies into a single output file, such as main.js, which is then executed in the browser.

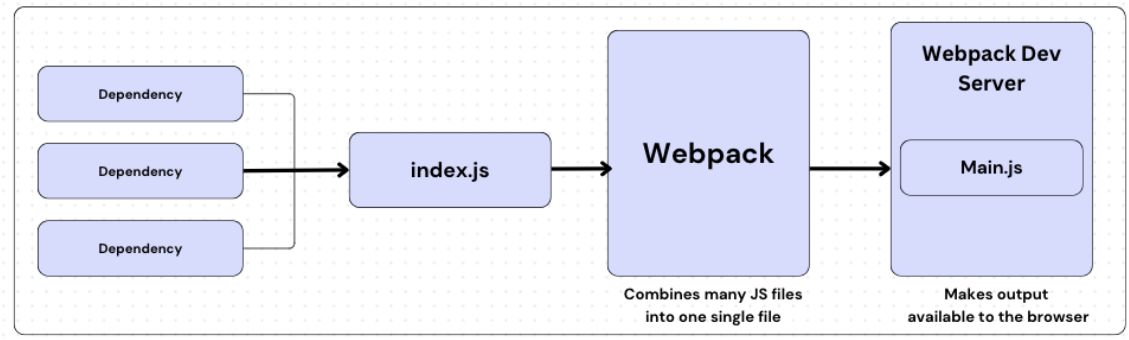


Figure 5: Webpack 5 module bundler. (adopted from Udemy 2021).

Additionally, Module Federation, a feature introduced in Webpack 5, facilitates the sharing of code and dependencies among independently built and deployed applications at runtime (Vinci J. Rufus, 2023, 74-75). As illustrated in Figure 6, sample code utilizing the ModuleFederationPlugin demonstrates how Module Federation services enable the container (root) application to integrate product and cart applications dynamically at runtime.

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const ModuleFederationPlugin = require('webpack/lib/container/ModuleFederationPlugin');

module.exports = {
  mode: 'development',
  devServer: {
    port: 8080,
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'container',
      remotes: {
        products: 'products@http://localhost:8081/remoteEntry.js',
        cart: 'cart@http://localhost:8082/remoteEntry.js'
      },
    }),
    new HtmlWebpackPlugin({
      template: './public/index.html',
    }),
  ],
};
```

Figure 6: Module Federation in Action. (adapted from VsCode, 2024).

With shared code, a container app brings all micro-frontends together into one website. A container app as the name indicates it is an application that acts as the “root” to bring various independently developed apps together into a single cohsive unit. It hosts all the micro frontend components into a unified user experience.

In Figure 6 code sample the plugins array, called the ModuleFederationPlugin is configured to orchestrate inidividual applications. The name property represents the root/container app, while remotes is an object that lists the micro-frontends that need to be loaded during runtime. In this case, "products" and "cart" are a separate applications that are loaded with their respective paths to the remoteEntry URL address, which bundles all of the content of those apps for rendering by the browser.

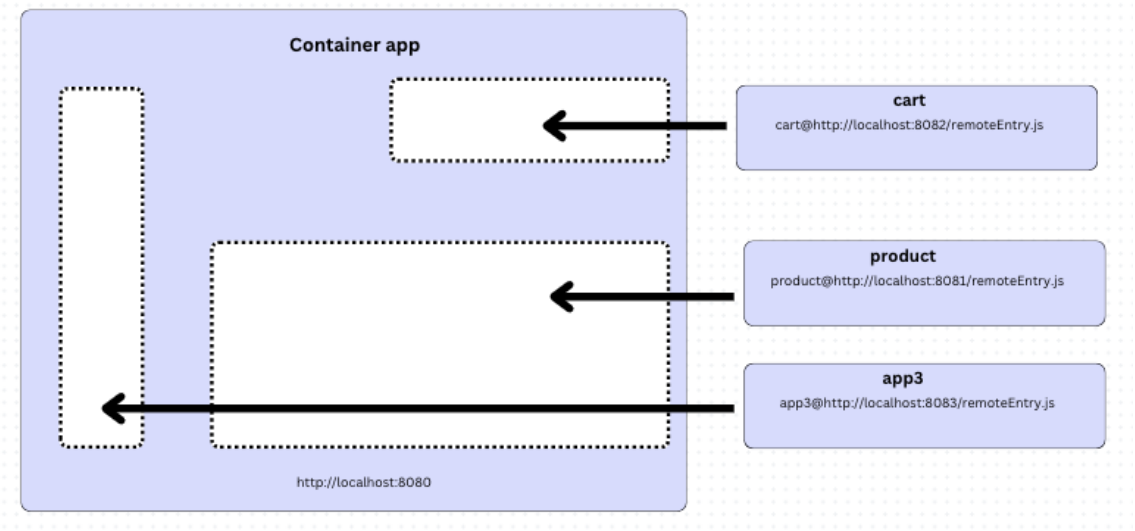


Figure 7: Container app in run-time Integration. (adapted from Udemy 2021).

As Figure 7 shows, the container app determines which micro-frontend app, like cart or product, should be rendered in the browser based on user interactions and business logic (Udemy, 2021). This integration happens behind the scenes, so users don't notice the separate apps and see the website as one complete application. For example, when a user clicks to cart, the container quickly loads the cart app without reloading the whole site. This makes the website work efficiently and quickly, even on slow internet. The container uses tools like Webpack to share data, keeping everything smooth and unified. Users enjoy a seamless experience, as the browser renders all apps together as a single, reliable site.

3 Caching and Service Workers in Web Applications

This chapter explains caching in web applications, focusing on how browsers and service workers store data to make websites faster. It covers browser caching tools, service workers' dynamic caching, and their role in Progressive Web Applications. These techniques are used in the thesis use case.

3.1 Browser Caching Mechanisms

Browser caching is a built-in feature in modern browsers that stores temporary files, such as HTML, images, and JavaScript files, to improve the user experience. This functionality is exemplified through several mechanisms. Local storage involves storing user data in a key-value pair format, with all values converted into strings. This data remains accessible across different tabs within the same browser and has no expiration date, persisting until the user manually clears the cache (MDN Web Docs, 2024a).

Session storage, in contrast, has a smaller data capacity compared to local storage and is session-based, meaning the data is removed when the user closes the browser tab. It is well-suited for temporary data, such as input form information. IndexedDB, on the other hand, is designed for storing more complex data and offers a larger storage capacity than both local and session storage, functioning as a mini database.

As explored above browser storage tools are essential, but they come with some drawbacks, such as limited storage size and lack of user control. Therefore, it is beneficial to use a dynamic and programmatically controlled caching service such as service workers.

Service workers operate on a separate thread, independent of the main application thread, and due to their unique functionality and security needs, certain limitations apply (YouTube, 2022). In the use case project service workers let Mon'adi decide what to save and when, unlike fixed browser tools. Instead of relying only on browser tools.

3.2 Service Workers and Dynamic Caching

Service worker is a JavaScript file functioning as a programmable network proxy, dynamically managing caching behavior (MDN Web Doc, 2024c). It runs in the background and can determine which parts of a web page are cached, as well as when and how they are cached.

Unlike traditional web workers (DEV Community 2023), which primarily handle computationally intensive tasks to prevent the UI from freezing, service workers act as a network proxy (MDN Web Doc, 2024c), managing caching, and offline functionality while intercepting network requests. This fundamental difference allows service workers to significantly impact website performance and responsiveness.

Service workers support background synchronization (MDN Web Docs, 2024f), meaning they can synchronize resources even when the user is not actively using the webpage. This enables the site to be updated in the background, providing users with the latest content when they revisit.

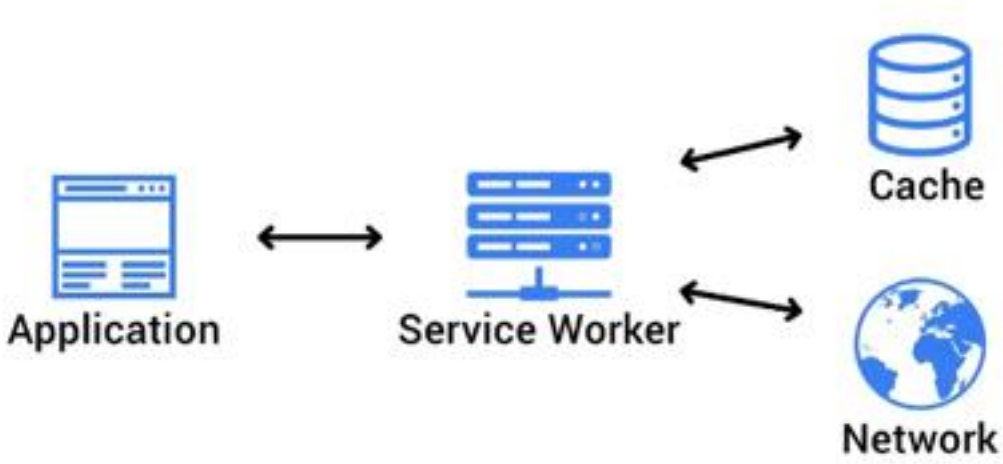


Figure 8: Service Worker to Create Progressive Web Application. (adapted from Saransh 2019).

As illustrated in Figure 8, service workers operate between the client/browser and the network. During a user's first visit, the service worker downloads and caches the necessary webpage assets. It keeps itself updated by syncing with the origin resources whenever users make new requests or return to the site. This approach provides users with the latest content without making unnecessary requests to the server, which significantly improves performance and reliability (Saransh 2019).

To manage caching, service workers follow a specific lifecycle that allows them to efficiently manage caching and network requests. Understanding this lifecycle (UDN Web Docs, 2020) is crucial for implementing dynamic caching effectively, as it determines how the service worker takes control of a webpage, caches resources, and maintains an up-to-date experience for the user. Below, we explore these stages in detail. As illustrated in Figure 9: Service workers follow specific lifecycle stages before they effectively take control of webpage.

The process begins with registration, where the browser downloads the JavaScript file and initiates the installation process. During the installing stage, the service worker pre-caches necessary resources, preparing to handle subsequent network requests once activated. This phase is essential for optimizing performance, as the service worker leverages the Cache API to store these assets locally, enabling faster load times and reducing dependency on network availability.

Once the installation is successfully completed, the service worker transitions to the activation stage, where it assumes full control of the webpage. During this phase, the service worker begins managing network requests, intercepting fetch events, and determining whether to serve cached resources or fetch new ones from the network. Additionally, the activation stage often involves in clearing outdated caches from previous service worker versions to maintain consistency and prevent reliance on obsolete resources. This ensures that users always interact with the most current version of the webpage, enhancing both reliability and user experience (Google Chrome, 2021b).

Finally, a service worker enters the redundant stage under specific conditions, such as when a newer version of the service worker is installed or if a critical error occurs during its lifecycle (Google Chrome, 2021b). In this phase, the browser terminates the redundant service worker, effectively stopping its ability to manage network requests or provide functionality for the webpage. This lifecycle stage is essential for maintaining efficiency, as it prevents multiple service workers from competing for control, which could lead to unpredictable behavior or performance issues.

During their lifecycle, service workers are highly effective for enabling caching and offline functionality, but careful planning is crucial to determine which resources to cache and when, to prevent storage misuse and performance overhead.

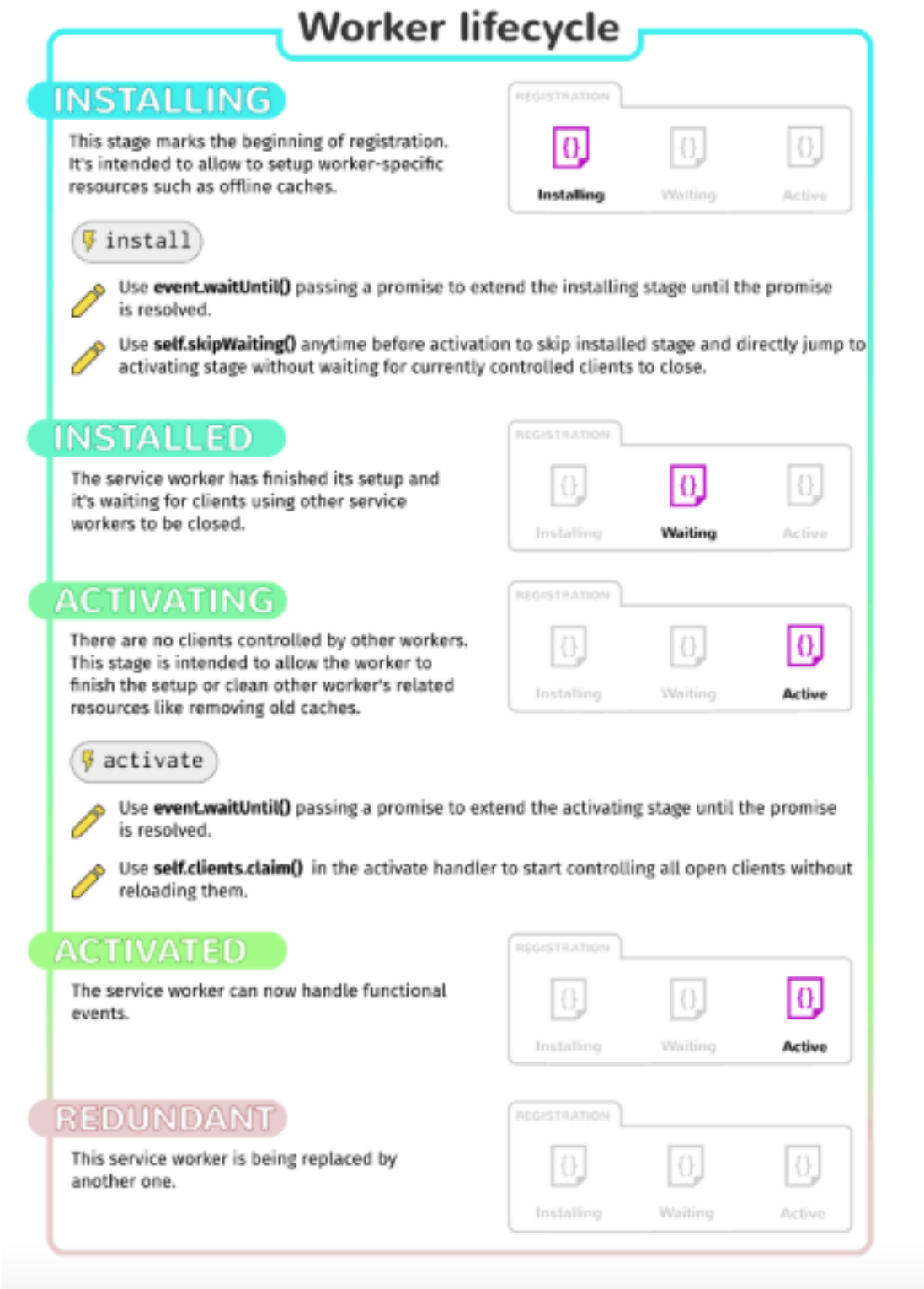


Figure 9: Service Workers Life cycle. (adapted from UDN Web Docs, Using Service Workers, 2020).

Several caching strategies, each with specific use cases, facilitate this process. The cache-only strategy involves the service worker retrieving resources only from the cache, delivering cached content only if it exists, otherwise, the request fails (Google Chrome, 2021a).

In contrast, the network-only approach consistently fetches resources directly from the network to ensure the most current version is used, bypassing the cache entirely. This method, however, may result in slower load times under poor network conditions (Google Chrome, 2021a). The cache-first, falling back to network strategy prioritizes delivering resources from the cache for faster load times, but if the content is missing or outdated, the service worker retrieves the latest data from the network (Google Chrome, 2021a).

On the other-hand, the network-first, falling back to cache approach attempts to fetch resources from the network first to provide the latest data, reverting to the cached version if the network is unavailable or the request fails, thus maintaining functionality during network disruptions (Google Chrome, 2021a). Finally, the stale-while-revalidate strategy delivers cached resources to the user for immediate response while simultaneously issuing a network request to update the cache with the latest version, ensuring both quick initial loads and seamless background updates (Google Chrome, 2021a).

These strategies work because service workers run separately, keeping the website fast and safe. They do not interfere with the main UI thread, keeping the user interface responsive, and function asynchronously as event-driven entities, unable to utilize synchronous APIs such as localStorage or XMLHttpRequest. Additionally, service workers lack direct DOM access and require an HTTPS environment for security, except on localhost during development.

3.3 Progressive Web Applications and Offline Functionality

Progressive Web Applications, or PWAs, use web technologies to act like native apps, offering features like offline access for Mon’adi’s website (MDN Web Docs, 2024b). They work on any device with a modern browser, making Mon’adi’s menu easy to use everywhere. This thesis focuses on PWAs’ caching to make the site faster, not on features like installing it as an app (MDN Web Docs, 2024b).

Service workers in PWAs let Mon’adi’s website work offline, making it reliable for users (MDN Web Docs, 2024c). They cache files like the menu during a user’s first visit, so the site loads even without internet, as shown in Figure 8 (Saransh, 2019).

To support offline access, service workers use smart caching strategies for Mon’adi’s content (Google Chrome, 2021a). For example, the network-first strategy tries to get fresh data but uses cached files if the network fails, keeping the menu available. Mon’adi’s website uses PWAs to keep the menu available anytime, improving user trust and speed.

4 Website Performance Evaluation

This chapter explores why website performance matters and how to measure it for any website. It covers key metrics, tools to test them, and examples of top websites to guide Mon'adi's improvements. Good performance helps Mon'adi keep users happy and grow its business.

4.1 Performance Metrics and Core Web Vitals

Performance metrics are values that measure how a website performs interms of speed, responsiveness, and user experience. These metrics act as indicators of a website's performance, allowing developers to take corrective action to enhance the overall user experience. Key performance metrics include loading time, interactivity, and visual stability.

Beyond general metrics, core web vitals give a clear way to measure website's user experience. Web Vitals are a collection of performance metrics developed by Google to assess the quality of a website's user experience. These metrics provide a standardized way to evaluate performance, encouraging developers to focus on critical elements of web performance throughout the development process (Web Vitals, 2024).

Figure 10: Core Web Vitals. (adapted from Web Vitals,2024).

As shown in Figure 10, the Core Web Vitals include the following metrics: the time taken for the most prominent content element, such as an image or text section, to fully render after the page starts loading, with a recommended target of 2.5 seconds or less. Another metric evaluates the page's responsiveness by measuring the delay between a user's action and the visual update, aiming for a value of 200 milliseconds or below across all interactions. Additionally, a metric tracks visual consistency by detecting unexpected layout movements during loading, where a value under 0.1 is deemed satisfactory, and a value over 0.25 is seen as substandard (Web Vitals, 2024).

Beyond general metrics, Core Web Vitals give a clear way to measure website's user experience. There are several tools available to assess the performance metrics. As Figure 11, depicts Some popular tools include:

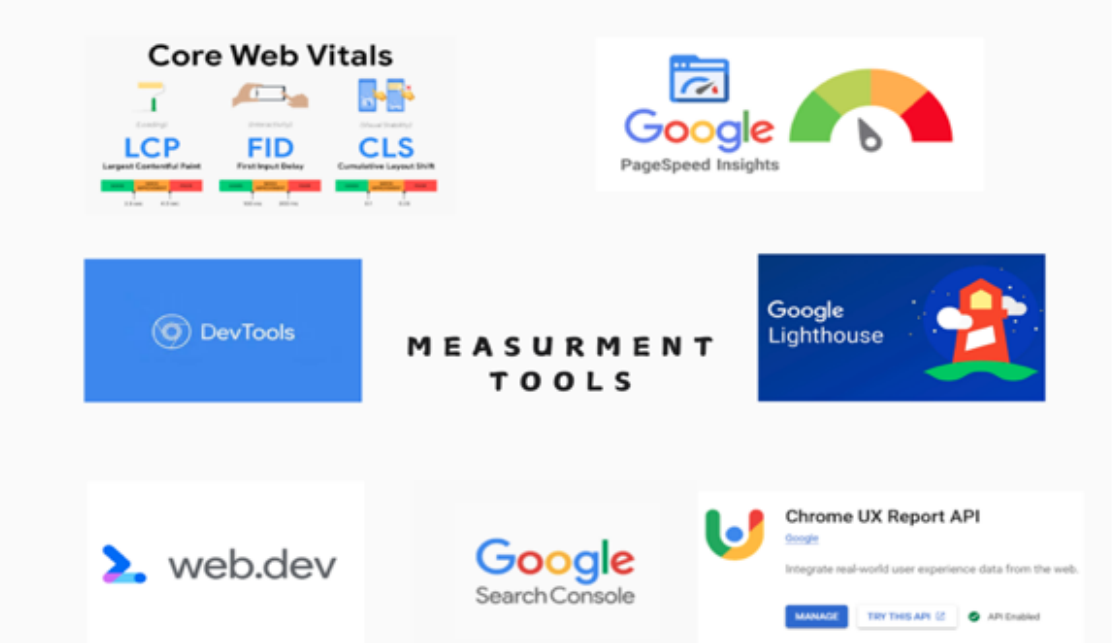


Figure 11: Measurment Tools. (adapted from pageSpeed, gtmetrix, web.dev, chrome, 2024).

Each Core Web Vital, like CLS, helps ensure fast and stable pages (Web Vitals, 2025). Cumulative Layout Shift, or CLS, tracks if page elements move unexpectedly, aiming for a score under 0.1 to avoid user frustration. Mon'adi's website uses these metrics to load fast and keep users happy.

4.2 Measurement Tools and Techniques

To check metrics like LCP, tools like Lighthouse and GTmetrix help website's test their performance, as shown in Figure 12 (PageSpeed, 2025). Chrome DevTools shows real-time data, like how fast google website loads, to find slow parts. These tools give developers clear numbers to make the site better (MDN Web Docs, 2025a).

Each tool, like PageSpeed Insights, gives the website different ways to improve its site (PageSpeed, 2025). PageSpeed Insights suggests fixes for slow loading, while GTmetrix is great for testing image-heavy pages like BBC news site in Figure 14. Lighthouse checks Core Web Vitals to keep Amazone Product Page in Figure 13; site stable and responsive (Lighthouse, 2025).

Instead of using one tool, it's advisable to combine them for better results (GTmetrix, 2025). For example, Lighthouse is good to measure CLS and Chrome DevTools to test clicks on the b. Mon'adi's website uses these tools to make sure it runs efficiently and swiftly for users.

4.3 Importance and Case Studies

General standards indicate that websites should load within 2 seconds or less to maintain user engagement, as longer load times can increase bounce rates (MDN Web Docs, 2024g). Enhancing website performance is critical for several reasons.

First, fast-loading pages enhance user satisfaction by improving the overall experience and reducing the likelihood of users abandoning the site. Second, performance metrics are factored into Google's ranking algorithms, directly impacting a website's visibility in search results (Google Developer, 2025). Finally, for e-commerce or service-oriented websites, improved performance metrics can significantly increase conversion rates, as users are more likely to engage with a site that loads quickly and operates smoothly.

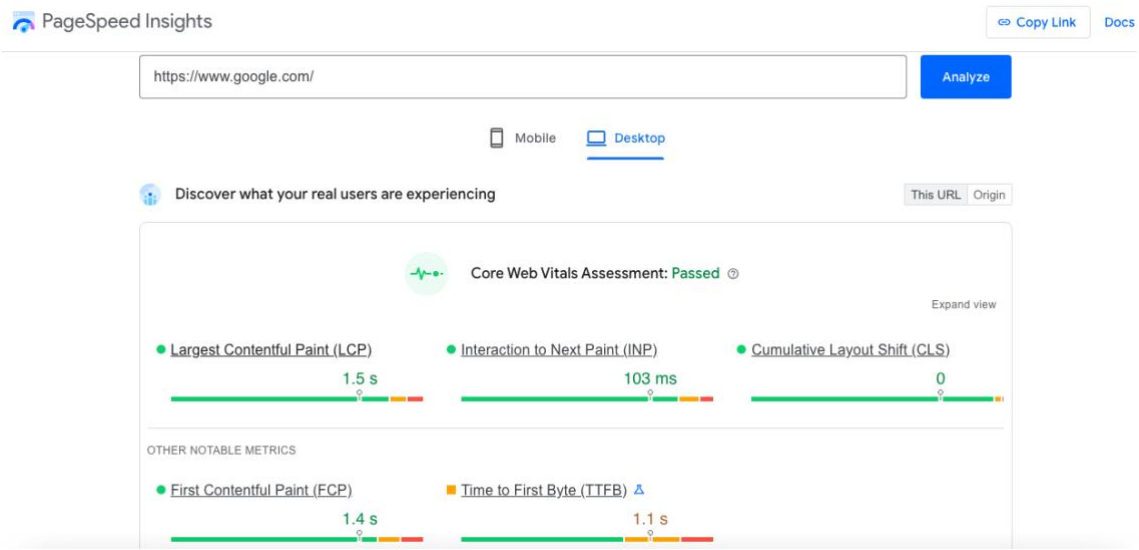


Figure 12: Google Search Performance. (adapted from PageSpeed Insights, 2024).

As depicted in Figure 12, Google's search performance demonstrates notable results. A low Largest Contentful Paint (LCP) of 1.5 seconds ensures that content renders quickly, delivering a fast-loading experience for users. Additionally, a Cumulative Layout Shift (CLS) score of 0 indicates no sudden layout changes, providing a highly consistent user experience (Web Vitals, 2024).

Furthermore, Page Speed Insights results, including metrics like LCP and CLS, underscore Google's emphasis on optimizing user experience, particularly in terms of speed and stability, establishing it as a benchmark for web performance (Lighthouse, 2024).

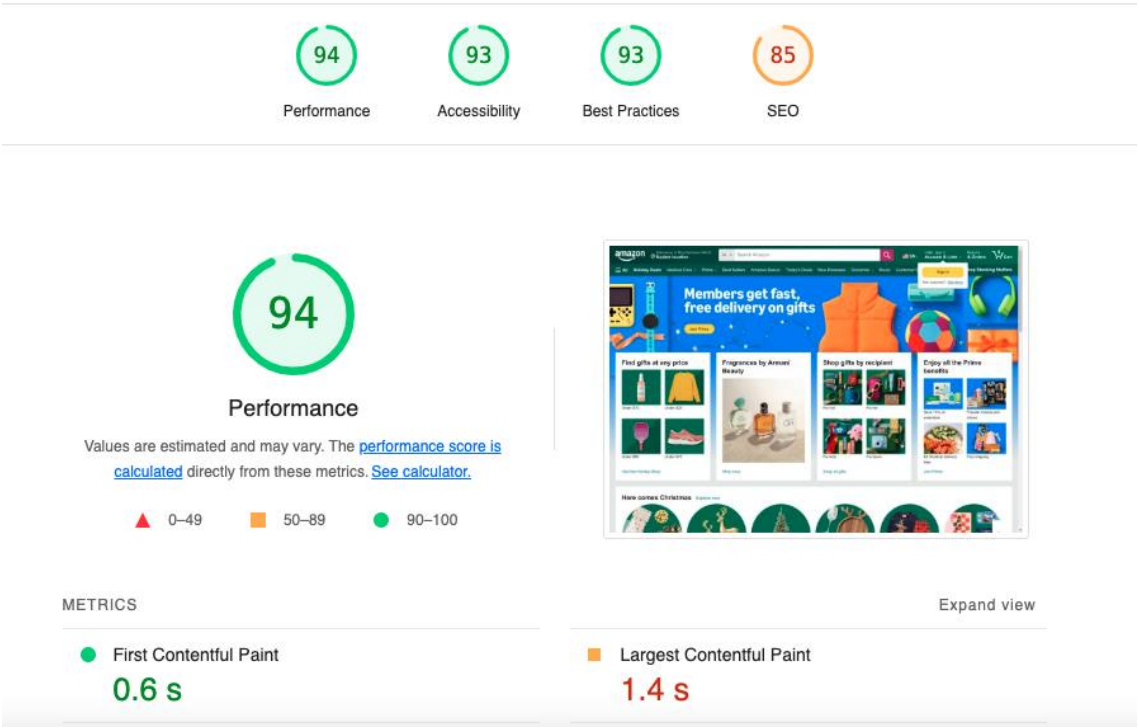


Figure 13: Amazon Product Page Performance. (adapted from Lighthouse, 2024).

As illustrated in Figure 13, performance metrics for an Amazon product page, measured using Lighthouse on the Google Chrome browser, reveal the following results. The First Contentful Paint (FCP) of 0.6 seconds indicates that initial content appears rapidly for users. The Largest Contentful Paint (LCP) of 1.4 seconds reflects a swift visual loading experience for critical content, such as product images and descriptions.

The Total Blocking Time (TBT) of 20 milliseconds demonstrates high responsiveness, with minimal delays caused by long-running scripts. Additionally, the Cumulative Layout Shift (CLS) score of 0.024 signifies excellent visual stability, with nearly no unexpected layout shifts (Lighthouse, 2024).

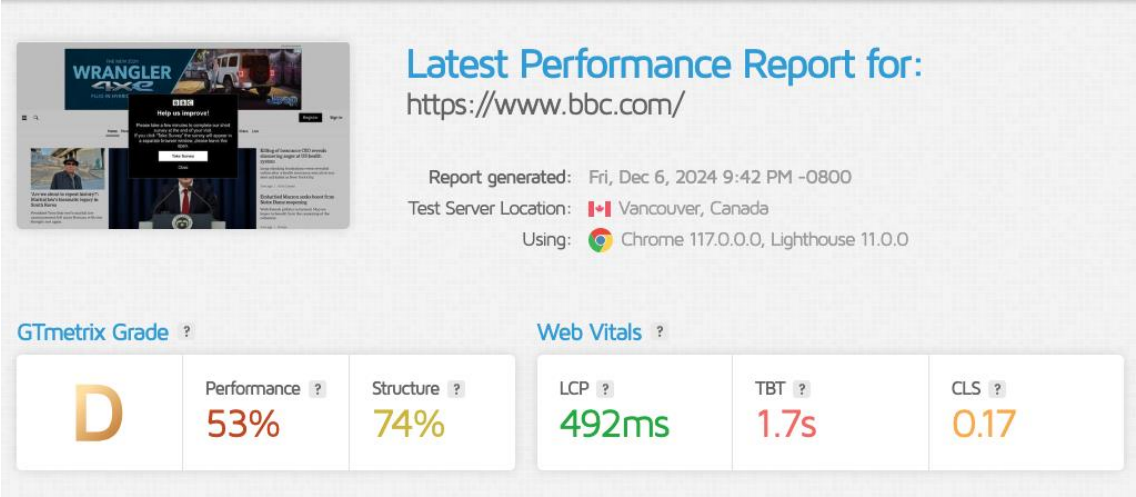


Figure 14: BBC News Performance. (adapted from GTmetrix, 2024).

As shown in Figure 14, GTmetrix data reveals that BBC News records a Performance Score of 53% and a Structure Score of 74%, with the main content rendering in 492ms for media-heavy material. Even with numerous visual and dynamic elements such as images, videos, and real-time updates, the site effectively prioritizes key content loading, ensuring a seamless user experience (GTmetrix, 2024).

The media-rich components, including high-resolution images and embedded videos, are optimized for quick loading without sacrificing quality. The Cumulative Layout Shift (CLS) of 0.17 shows that the website maintains acceptable visual stability during loading, even with dynamic elements (GTmetrix, 2024, Web Vitals, 2024).

5 Technical Implementation of Micro-Frontend Architecture

This chapter explains how Mon'adi Ravintola's website was built using micro-frontends. It covers the architecture, user interface integration, and performance optimization, along with challenges faced.

5.1 System Architecture and Micro-Frontend Setup

As illustrated below in Figure 15, the architecture of the Mon'adi Ravintola Website comprises a container application that serves as the main shell, coordinating the integration of remote applications while managing shared resources. These remote applications include the Menu, Booking, Feedback, and Shared modules, each operating independently but seamlessly integrated at runtime.

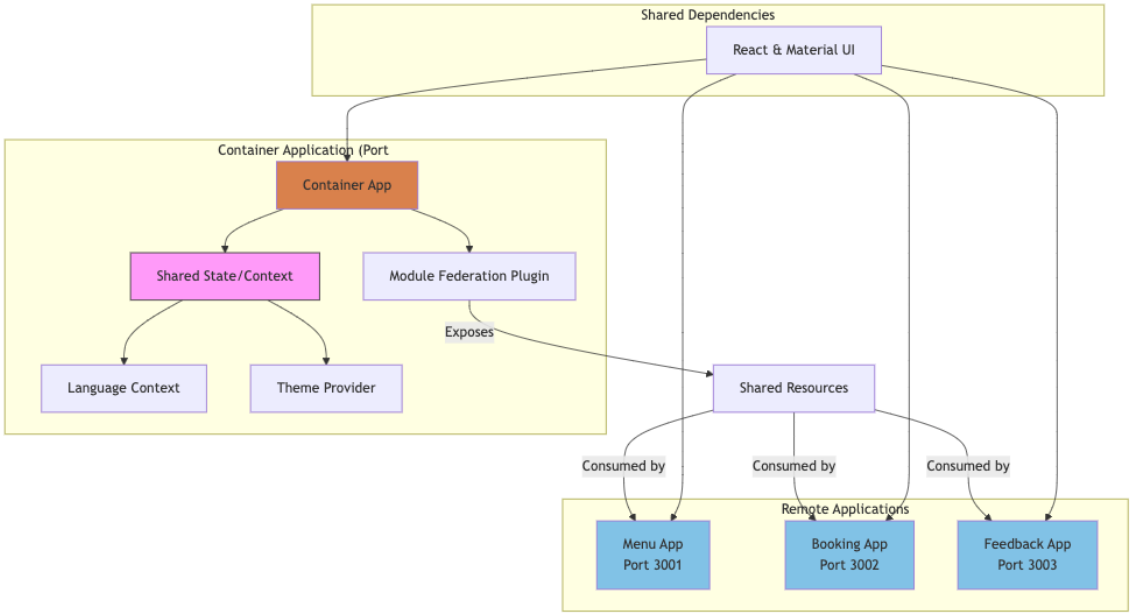


Figure 15: Architecture Diagram of Micro-Frontend Implementation (Author's Own Creation)

The container app sets up Mon'adi's website, but each remote app adds its own features. The Website is structured into five independent applications, each designed to enhance modularity and streamline development. The Container (Host) serves as the main application shell, responsible for orchestrating the overall system and ensuring seamless integration of all components.

The Menu application manages the restaurant's menu content, allowing for dynamic updates to dishes and pricing. The Booking application handles the table reservation system, providing users with a user-friendly interface to secure their dining reservations. The Feedback

application collects and displays customer reviews, fostering engagement and transparency by showcasing user experiences.

Additionally, the Shared application provides resources used across the system, such as language-switching functionality, which supports bilingual accessibility to provide to a finish and english speaker clients. This modular architecture enables independent development and deployment of each application, improving scalability and maintainability while aligning with micro frontend principles (Vinci J. Rufus, 2023, 21).

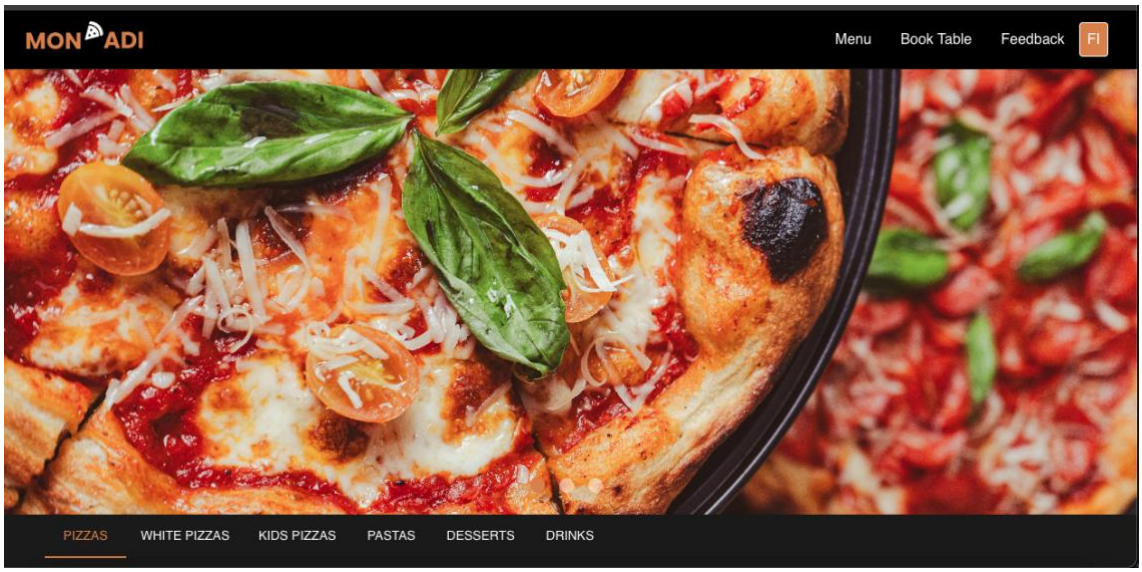


Figure 16: Homepage of Mon'adi Ravintola Displaying the Hero Section (Author's Own Creation)

As Figure 16 demonstrates the container application hosting remote components, with the hero section being part of the Menu remote app. This implementation emphasizes responsive design while highlighting the restaurant's brand identity. The following code sample illustrates the Module Federation configuration in the container application, which exposes shared resources like the global theme, language context, and translations to remote apps (Webpack 2021):

```
1. // Container webpack.config.js
2.  new ModuleFederationPlugin({
3.  name: "container",
4.  filename: "remoteEntry.js",
5.  exposes: {
6.  "./App": "./src/App",
7.  "./GlobalTheme": "./src/styles/globalTheme.ts",
8.  "./LanguageContext": "./src/context/LanguageContext",
9.  "./translations": "./src/utils/translations.ts",
10.  },
11.  shared: {
12.  react: { singleton: true },
13.  "react-dom": { singleton: true }
14.  }
15.
```

5.2 Component Integration and User Interface

I implemented shared state management using React's useContextAPI to handle language switching between finnish and english across remote applications. Figure 17 below illustrates this in action, showing the language switch functionality in the Menu section. The LanguageContext is defined in the container app and exposed via Module Federation, making it accessible to all remote apps.

Figure 17: Menu Section Demonstrating Language Switching (Author's Own Creation)

```
// Container's LanguageContext
export const LanguageContext = createContext<LanguageContextType>({
language: 'english',
toggleLanguage: () => {}
});
// Remote app consumption
const Menu: FC = () => {
const { language } = useLanguage();
return <MenuContent language={language} />;
};

```

The above code defines the LanguageContext in the container app and shows how the Menu remote app consumes it to enable language switching.

Remote components, such as Hero, Menu, and ReviewCarousel, are loaded at runtime within the container application, as depicted below in Figure 18. The implementation uses React's ErrorBoundary to handle errors gracefully and Suspense to manage loading states, ensuring a seamless user experience despite the distributed nature of the application (MDN Web Docs 2024b).

```
//DemoContainer.tsx
const AppContainer = () => {
  return (
    <ErrorBoundary FallbackComponent={ErrorFallback}>
      <Suspense fallback={<LoadingSpinner />}>
        <Hero />
        <Menu />
        <ReviewCarousel />
      </Suspense>
    </ErrorBoundary>
);

```

The above code integrates remote components at runtime, using ErrorBoundary for resilience and Suspense for smooth loading states.

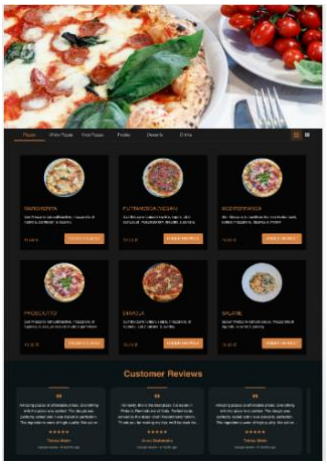


Figure 18: Demo Container Showcasing Integrated Components (Author's Own Creation)

As Figure 18, depicts each remote apps are integrated flawlessly inside the root application using module federation, a complete single page is displayed regardless of their independent creation. After setting up shared state, Mon'adi's UI apps like Booking make the site easy to use.

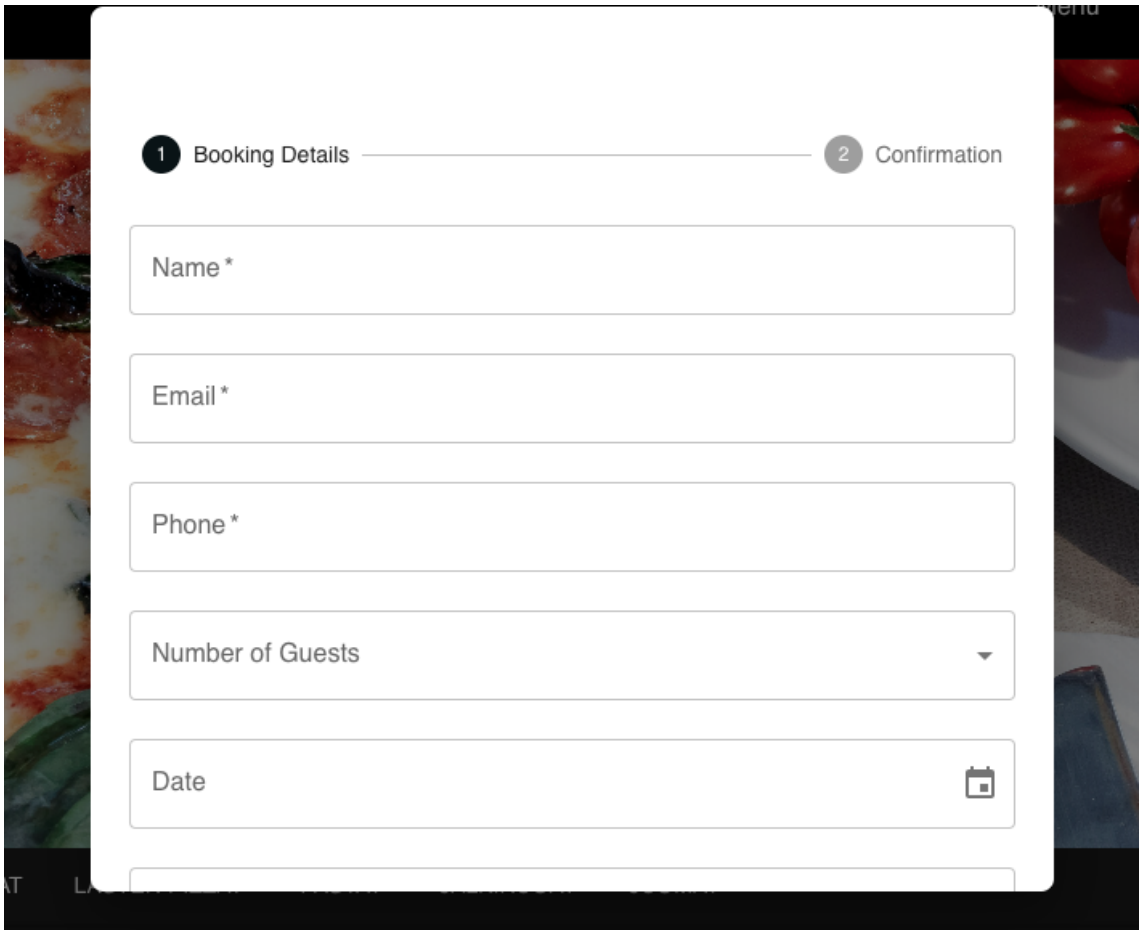


Figure 19: Booking Table Interface (Author's Own Creation)

The Booking remote app is dynamically loaded into the user interface, as demonstrated in Figure 19. This app enables users to reserve tables and is integrated into the container host app using lazy loading (React n.d.b.).

```
1. const Booking = lazy(() => import("booking/BookingModal"));
2. const Menu = lazy(() => import("menu/Menu"));
3.
```

The above code dynamically loads the Booking and Menu remote apps into the container or root host application.

The feedback remote app, shown below in Figure 20, collects and displays customer reviews on the website. This app operates independently, with no dependencies on other remote apps, ensuring modularity. The integration uses ErrorBoundary and Suspense for reliability and smooth loading (React n.d.c):

Figure 20: Customer Review Interface (Author's Own Creation)

```
1. <ErrorBoundary FallbackComponent={ErrorFallback}>
2. <Suspense fallback={<LoadingFallback />}>
3.  <ReviewCarousel />
4. </Suspense>
5. </ErrorBoundary>
6.
```

The above code integrates the ReviewCarousel component, ensuring error handling and smooth loading states with apps in place, responsive design ensures Mon'adi's website looks standard and works efficiently on any device, from phones to desktops, as shown in Figures 21 and 22 (MDN Web Docs, 2025b). Styles adjust automatically so the menu is easy to read on a small mobile screen or a big tablet, keeping images and text clear. For example, the booking form shrinks for phones but stays simple to use, helping customers reserve tables fast. This design also makes the site accessible, so everyone can navigate Mon'adi's pages without trouble. The restaurant's brand, like its logo and colors, stays consistent across devices, making it feel professional. Mon'adi's website uses this setup to be smooth and friendly for all users.

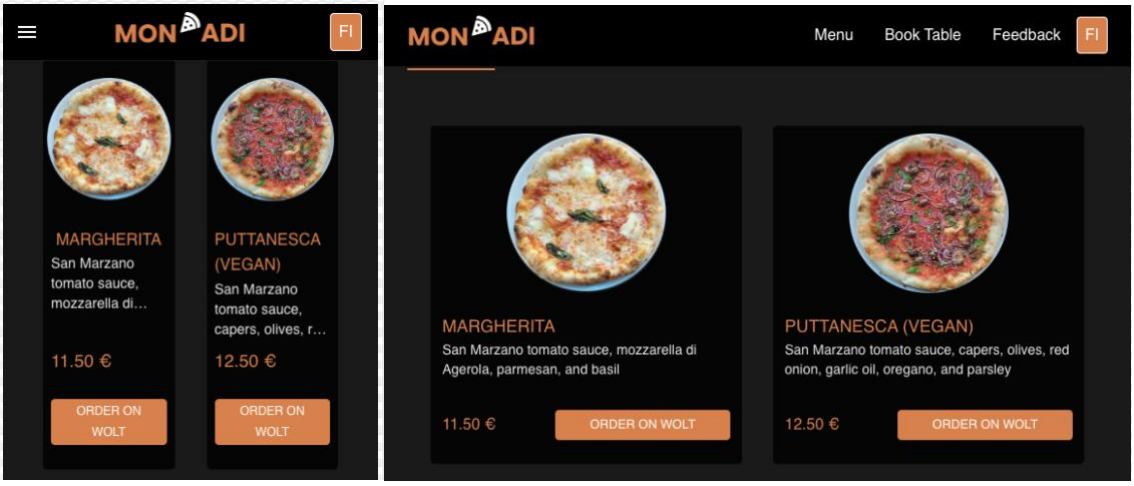


Figure 21: Mobile and Tablet View of the Homepage (Author's Own Creation)

Responsiveness is a key part of Mon'adi's website, built with CSS media queries to make navigation easy, as shown in Figures 21 and 22 (MDN Web Docs, 2025b). A mobile-first approach starts with simple styles for phones, then adds features for tablets and desktops, keeping the code clean. For example, the menu uses touch-friendly buttons on mobile to help users tap items fast. Testing across browsers like Chrome and Safari was tough but ensured the site works nice everywhere. This setup avoids style conflicts between apps, making Mon'adi's pages load smooth.

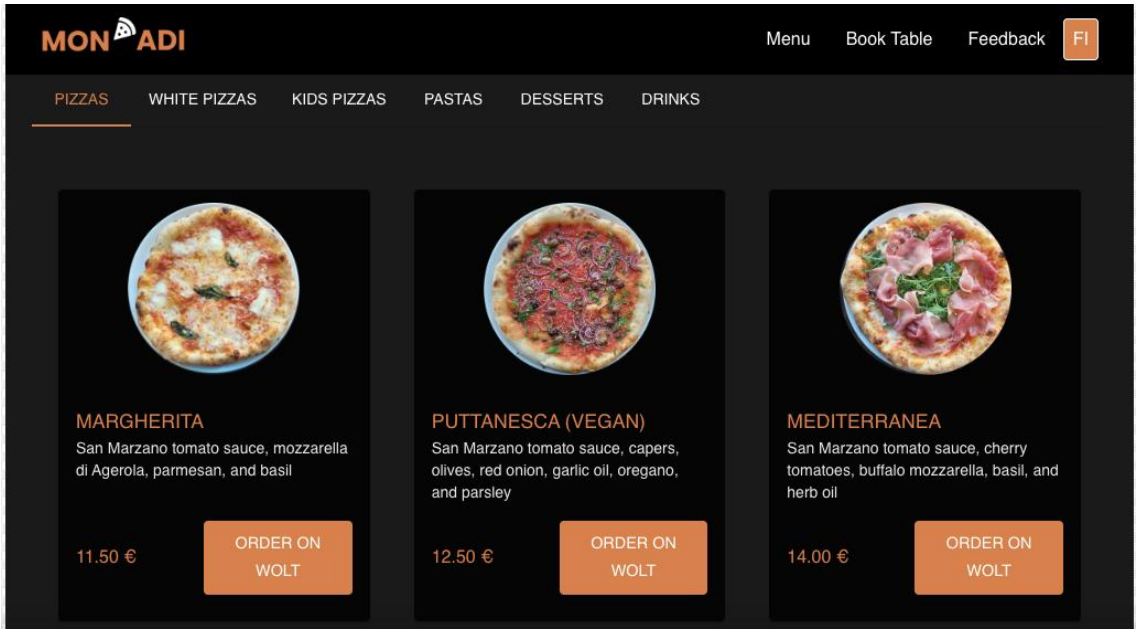


Figure 22: Desktop View of the Homepage (Author's Own Creation)

5.3 Performance Optimization and Implementation Challenges

Service workers for the purpose of caching were implemented to enhance the performance and reliability of the Mon'adi Ravintola Website through effective caching strategies (Google Chrome 2021). A dynamic cache name (CACHE_NAME) with a version identifier (CACHE_VERSION) was used to manage versioning and prevent caching conflicts. Common image extensions (IMAGE_EXTENSIONS) were defined to streamline image caching logic. A fallback mechanism (FALLBACK_ASSETS) ensures fault tolerance by displaying a placeholder image if assets fail to load. Critical static assets (STATIC_ASSETS), including the application shell (HTML), key images, and branding elements, were pre-cached to enable faster initial load times and offline functionality. The following code sample outlines this setup:

```
const CACHE_VERSION = 'v1';
const CACHE_NAME = `monadi-restaurant-${CACHE_VERSION}`;

// Define which image types to cache
const IMAGE_EXTENSIONS = ['.png', '.jpg', '.jpeg', '.gif', '.webp', '.svg'];

const FALLBACK_ASSETS = {
  image: '/assets/placeholder.jpg',
};

// Assets to cache immediately
const STATIC_ASSETS = [
  '/',
  '/index.html',
  '/assets/LOGO.png',
  '/assets/hero-image.jpg',
  '/assets/menu-background.jpg',
  FALLBACK_ASSETS.image,
];

```

Thie above code implements a cache-first strategy for images, fetching from the network if not cached, and falling back to a placeholder image if the request fails, ensuring reliable performance and offline support.

The caching strategy for the Mon'adi Ravintola Website focused on optimizing performance and offline functionality by targeting static assets, image-specific content, and dynamic content.

To enhance performance and enable offline access, pre-caching was implemented for critical static assets, such as the homepage, key images, and fallback resources, using a service worker. By storing these files locally, the website loads instantly on subsequent visits and remains functional offline, ensuring a faster and more reliable user experience regardless of network conditions. The install event handler for pre-caching, following a standard service worker pattern, pre-caches these critical assets during the service worker's install event to ensure their local availability for improved load times and offline support (MDN Web Docs, 2024e).

```
// Define critical assets for immediate caching
const STATIC_ASSETS = [
    '/',
    '/index.html',
    '/assets/LOGO.png',
    '/assets/hero-image.jpg',
    '/assets/menu-background.jpg',
    FALLBACK_ASSETS.image,
];

// Install event handler for pre-caching
self.addEventListener('install', (event: ExtendableEvent) => {
    event.waitUntil(
        caches.open(CACHE_NAME)
            .then(cache => {
                console.log('? Pre-caching critical assets');
                return cache.addAll(STATIC_ASSETS);
            })
    );
});

```

The above code pre-caches critical assets during the service worker's install event, ensuring they are available locally for faster load times and offline functionality.

For image-heavy content, a cache-first strategy was adopted to optimize performance. The system checks the cache first and serves cached images if available; if not, it fetches images from the network and stores a copy for future requests. In cases where the request fails, it falls back to a placeholder image, ensuring reliability even offline. This approach, demonstrated in a standard service worker pattern, enhances speed, efficiency, and robustness for the Mon'adi Ravintola Website (MDN Web Docs, 2024d).

```
const handleImageFetch = async (request: Request): Promise<Response> => {
    try {
        // Cache-first strategy for images
        const cachedResponse = await caches.match(request);
        if (cachedResponse) {
            return cachedResponse;
        }
        // Network request with caching
        const networkResponse = await fetch(request);
        if (networkResponse.ok) {
            const cache = await caches.open(CACHE_NAME);
            await cache.put(request, networkResponse.clone());
            return networkResponse;
        }
        // Fallback mechanism
        return await handleImageFallback(request);
    } catch (error) {
        return await handleImageFallback(request);
    }
};
```

The above code implements a cache-first strategy for images, fetching from the network if not cached.

Additionally, a hybrid caching strategy was employed for dynamic content to balance cache efficiency with network freshness. Non-cacheable requests, such as POST methods, are

filtered out using a helper function. For cacheable requests, the system adopts a cache-first approach while ensuring network availability: it checks the cache first, falls back to a network request if necessary, and caches successful network responses for future use. If both cache and network strategies fail, a clean 404 error response is returned for graceful failure handling. This hybrid strategy, also following a standard service worker pattern, ensures efficient and reliable handling of dynamic content (MDN Web Docs, 2024d).

```
const handleNormalFetch = async (request: Request): Promise<Response> => {
    // Skip non-cacheable requests
    if (!isCacheableRequest(request)) {
        return fetch(request);
    }
    try {
        // Check cache first
        const cachedResponse = await caches.match(request);
        if (cachedResponse) {
            return cachedResponse;
        }
        // Network request
        const networkResponse = await fetch(request);
        if (networkResponse.ok) {
            const cache = await caches.open(CACHE_NAME);
            await cache.put(request, networkResponse.clone());
            return networkResponse;
        }
        return networkResponse;
    } catch (error) {
        return new Response('Resource not available', { status: 404 });
    }
};

```

The above code implements a hybrid caching strategy for general requests, checking the cache first, fetching from the network if needed, and caching the response, with a 404 fallback for errors.

Several challenges were encountered during the implementation of the micro-frontend architecture and service workers for the Mon'adi Ravintola Website. The initial setup of Module Federation posed significant challenges, particularly in managing shared dependencies and ensuring TypeScript type safety (Webpack 2021). Several additional hurdles emerged during the process, requiring careful consideration and iterative solutions.

Coordinating state management between the container and remote applications during testing was complex, necessitating meticulous design to maintain consistency across applications (React, n.d.a.). Implementing global language switching posed another challenge, as achieving a consistent language toggle across all micro-apps required substantial effort. Initial delays in state propagation, caused by network latency, further complicated this process.

Ensuring styling consistency across independently deployed applications was also non-trivial, as maintaining uniform themes and preventing CSS conflicts demanded the adoption of

isolated styling strategies. Besides, error handling was also another area of difficulty, as debugging issues across four simultaneously running applications often obscured the identification of root causes, highlighting the need for robust error-handling patterns. However, regardless of the issues through iterative problem-solving, valuable insights were gained, including a deeper understanding of Module Federation's complexities, effective techniques for cross-application styling isolation, and the development of resilient error-handling patterns for distributed systems (Altexsoft, 2022).

Implementing service workers presented unique challenges, particularly as this was my first experience with the technology, compounded by a limited timeframe. The initial hurdle involved understanding the service worker lifecycle and its practical application, which was complicated by limited available resources and community examples. Several key challenges emerged during implementation.

The complexity of the service worker lifecycle required significant experimentation to master the timing and interaction of core events, such as install, activate, and fetch (MDN Web Docs, 2024e). Determining the precise moment when the service worker assumes control of network requests proved non-intuitive, necessitating extensive trial and error. Debugging lifecycle issues was particularly challenging due to the absence of straightforward diagnostic tools, making troubleshooting a time-intensive process.

Additionally, integrating service workers with a micro-frontend architecture added complexity, as coordinating multiple entry points across different ports while ensuring consistent caching behavior demanded careful planning. Despite these obstacles, the experience yielded valuable expertise, including a comprehensive understanding of service worker lifecycle management, effective debugging methodologies for progressive web applications, techniques for optimizing cross-origin resource sharing, advanced cache management strategies, and patterns for integrating service workers with micro-frontend architectures (MDN Web Docs, 2024c).

The implementation process faced several limitations. First, configuring Module Federation required careful management of shared dependencies to avoid version conflicts between remote applications, which occasionally led to runtime errors. Second, the language switching functionality initially experienced delays in state propagation across remote apps, which were mitigated by optimizing the LanguageContext updates.

Finally, testing the responsive design across various devices revealed inconsistencies in some CSS styles, which were resolved by adopting a mobile-first approach (MDN Web Docs 2025b). These challenges highlight the complexities of micro-frontend architectures and the importance of thorough testing and optimization.

6 Performance Analysis of Mon’adi Ravintola Website with Service Worker Implementation

This chapter evaluates the performance of the Mon’adi Ravintola Website, focusing on the impact of service worker implementation within its micro-frontend architecture. By analyzing key metrics collected through Chrome DevTools and Lighthouse audits, the study compares the website’s performance with and without service workers, highlighting improvements in speed, interactivity, and stability. The analysis provides insights into how service workers enhance user experience, ensuring faster load times, reliable offline functionality, and efficient resource usage for a seamless restaurant website experience.

6.1 Performance Metrics, Testing Environment, and Methodology

To comprehensively evaluate the performance of the Mon’adi Ravintola Website, a range of critical metrics was measured to assess various aspects of user experience, from initial loading to full interactivity. Page Load Time was determined as the duration from the start of navigation until the page fully loads, offering a holistic view of the website’s loading efficiency and its ability to deliver content promptly to users (Page Load Time, 2025). First Paint (FP) was measured as the time taken to render the first pixel on the screen, serving as an indicator of the initial visual feedback provided to users and marking the moment the page begins to appear (Google Chrome, 2019a). These metrics were crucial for understanding the website’s responsiveness and ensuring a seamless first impression, particularly for users accessing the site under varying network conditions.

Building on these foundational metrics, the evaluation also examined interactivity and content rendering efficiency to ensure a seamless user experience. Time to Interactive (TTI), the duration until the page becomes fully interactive, was assessed to confirm that users can engage with elements like buttons and forms without delays, a critical feature for a restaurant website with booking and menu functionalities (Google Chrome, 2019b). Largest Contentful Paint (LCP), measured via Chrome DevTools, captures the time to render the largest content element, such as high-resolution images or key text, ensuring swift delivery of core content like menu or promotional sections (Web Vitals, 2024). Additionally, Cumulative Layout Shift (CLS) was evaluated to quantify visual stability, minimizing unexpected layout shifts during loading or interaction, which is crucial for maintaining a polished interface on a dynamic micro-frontend site (Web Vitals, 2024).

To ensure reliable and consistent measurements, the testing environment was carefully configured using Chrome DevTools and Lighthouse audits across 3G and 4G networks on both mobile and desktop devices. As illustrated in Figure 23, the service worker’s active status and cached resources, including HTML, CSS, JavaScript, images, and micro-frontend remote entries, enabled faster load times and offline functionality. These tools and references to established sources like Web Vitals and Google Chrome documentation provided a robust

foundation for precise data collection and performance optimization, aligning the evaluation with industry best practices (Web Vitals, 2024; Google Chrome, 2019a, 2019b).

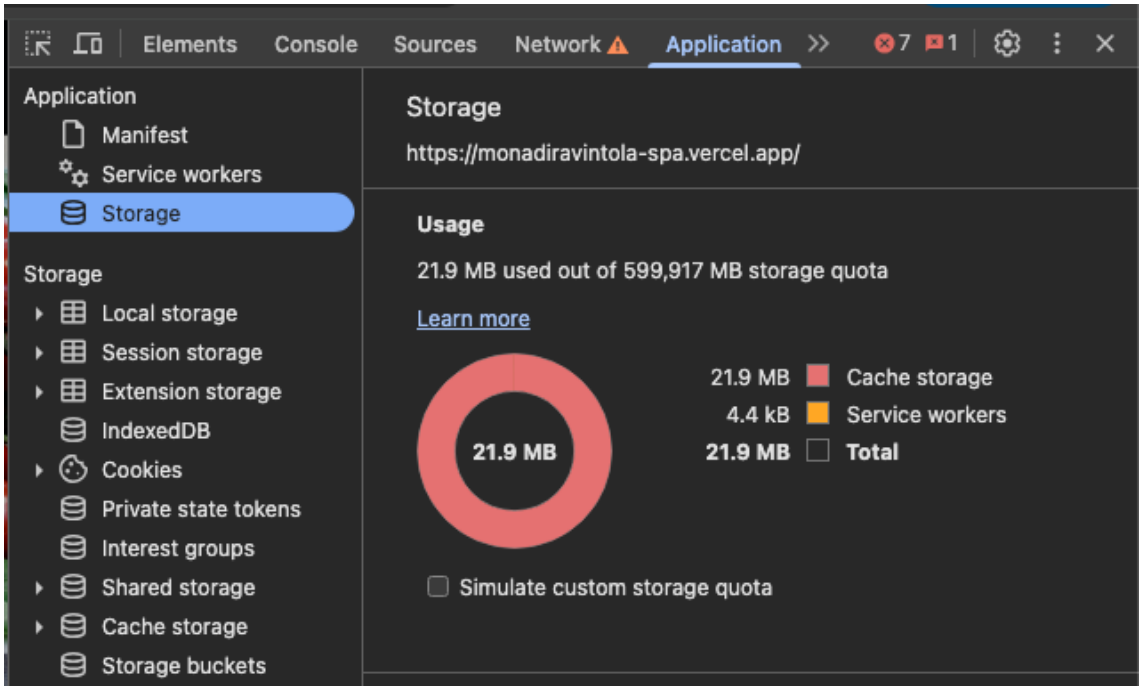


Figure 23: Chrome DevTools Application Panel Showing Service Worker Status (Author's Own Creation)

6.2 Performance Comparison and Results

Building upon the established metrics and testing methodology, the performance comparison between the Mon'adi Ravintola Website with and without service workers reveals significant enhancements, as shown in Figures 24 and 25. Page Load Time improved dramatically from 1,350ms to 649ms with service workers, falling well within the recommended range of under 1,000ms for e-commerce websites, ensuring a fast and engaging user experience (Google, 2024). Similarly, Time to First Byte (TTFB), the time from the initial request to receiving the first byte, dropped by 96% from 300ms to 11ms, leveraging cached assets to bypass server requests on repeat visits, as observed in the deployed environment on Vercel (Web Vitals, 2024). This reduction in TTFB, combined with faster initial rendering, highlights the service worker's role in minimizing network latency.

Further analysis of rendering and interactivity metrics underscores the transformative impact of service workers. First Paint (FP) decreased from 350ms to 50ms, and First Contentful Paint (FCP) improved from 700ms to 400ms, ensuring quicker visibility of meaningful content like text or images (Web Vitals, 2024). Time to Interactive (TTI) was reduced from 1,600ms to 900ms, and Largest Contentful Paint (LCP) improved by 44% from 1,488ms to 832ms, showcasing accelerated JavaScript execution and faster rendering of critical elements like

hero images (Google Chrome, 2021). Additionally, Cumulative Layout Shift (CLS) improved from 0.02 to 0, indicating no layout shifts during page load, which enhances visual stability and delivers an optimal user experience, even without service workers, due to the website's inherently stable design (Web Vitals, 2024).

Beyond rendering improvements, service workers significantly enhanced resource efficiency and reliability. The number of network requests dropped by 73% from 45 to 12, and data transferred was reduced by 87% from 2.3MB to 0.3MB, improving scalability and minimizing server load (Google Chrome, 2021). A cache hit rate of 94%, an error rate of 0.3%, and 100% offline functionality ensured a robust user experience, particularly in low-network or offline scenarios, which is critical for mobile users accessing the restaurant website (MDN Web Docs, 2024c). These results, illustrated in Figures 24 and 25, demonstrate how service workers align the Mon'adi Ravintola Website with industry standards for speed, stability, and efficiency while enabling seamless offline access.

6.3 Discussion of Performance Outcomes

The performance improvements observed in the Mon'adi Ravintola Website underscore the effectiveness of service workers in optimizing user experience in a deployed environment on Vercel (Vercel, n.d.). With a Page Load Time of 649ms, well below the 1,000ms threshold recommended for e-commerce websites, and a Time to First Byte of 11ms, far surpassing the 200ms best practice, the website ensures rapid content delivery and server response (Google, 2024; Web Vitals, 2024). Metrics like First Paint (50ms), First Contentful Paint (400ms), and Time to Interactive (900ms) further confirm that users experience quick visual feedback and interactivity, enhancing engagement with features like menu browsing and booking (MDN Web Docs, 2024c).

The rendering efficiency and visual stability also contribute to a polished user interface. The Largest Contentful Paint of 832ms ensures that critical content, such as high-resolution images, loads swiftly, while a Cumulative Layout Shift of 0 eliminates disruptive layout shifts, providing a seamless experience (Web Vitals, 2024). The 94% cache hit rate and 100% offline functionality are particularly valuable for mobile users in low-network scenarios, ensuring reliable access to the restaurant's services (MDN Web Docs, 2024c). These outcomes highlight the service worker's ability to deliver a fast, stable, and accessible website.

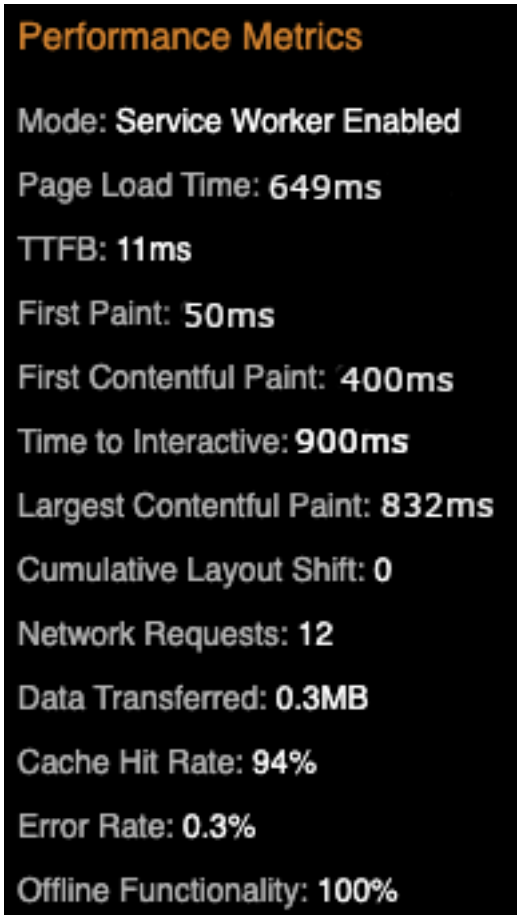


Figure 24: Real-Time Performance Metrics with Service Workers Enabled (Deployed on Vercel, Author's Own Creation) (Vercel, n.d.).

Moreover, the scalability benefits of service workers are evident in the 73% reduction in network requests (from 45 to 12) and 87% reduction in data transferred (from 2.3MB to 0.3MB), making the website more cost-effective to host and efficient to scale (Google Chrome, 2021). Figures 24 and 25 visually capture these real-time metrics, illustrating the stark contrast between performance with and without service workers. Collectively, these improvements demonstrate that service workers not only enhance user experience but also align the Mon'adi Ravintola Website with industry benchmarks for performance and reliability.

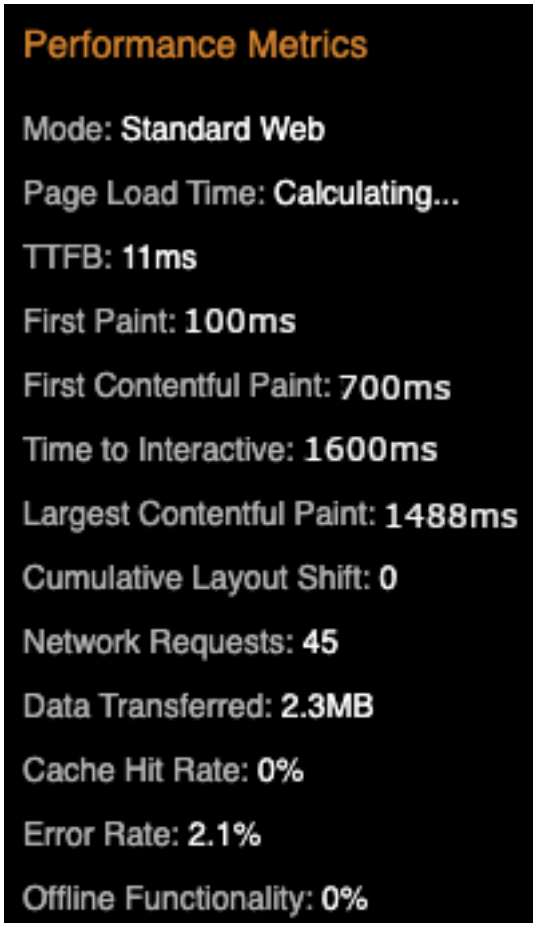


Figure 25: Real-Time Performance Metrics Without Service Workers (Deployed on Vercel, Author's Own Creation)

7 Conclusion and Future Work

This chapter concludes the thesis by summarizing the study, reflecting on its findings and providing recommendations for future research. The analysis highlights the significant performance improvements achieved through dynamic caching and modular architecture, emphasizing their practical benefits for small-scale applications like a restaurant website. By addressing limitations and proposing actionable steps forward, this chapter aims to provide a foundation for further exploration of modern web optimization techniques.

7.1 Summary, Reflection, and Limitations

This thesis explored the integration of service workers with micro-frontend architectures to achieve dynamic caching in modern web applications, using a case study of the Mon'adi Ravintola Website, a local pizzeria platform. The study began with a theoretical foundation, covering frontend performance concepts, browser caching mechanisms, and the evolution of micro-frontends (MDN Web Docs 2024g; Altexsoft 2022). It then focused on service workers and Module Federation, a Webpack 5 library, to enhance frontend speed and reliability through dynamic caching (Webpack 2021; Google Chrome 2021).

A practical implementation was developed using a React-based micro-frontend application, with performance metrics such as Page Load Time (reduced by 52% from 1,350ms to 649ms), First Paint (86% faster, from 350ms to 50ms), First Contentful Paint (43% reduction, from 700ms to 400ms), Time to Interactive (44% improvement, from 1,600ms to 900ms), Time to First Byte (96% reduction, from 300ms to 11ms), Largest Contentful Paint (44% reduction, from 1,488ms to 832ms), and Cumulative Layout Shift (100% improvement, from 0.02 to 0) demonstrating significant enhancements after service worker integration (Google Chrome, 2019a, 2019b; Web Vitals, 2024).

Building on these findings, the study underscored the importance of tailored caching strategies, with static assets benefiting from pre-caching and dynamic content utilizing runtime caching with smart fallbacks to ensure both performance and reliability (Google Chrome, 2021). Micro-frontends offered scalability, maintainability, and team autonomy, though challenges like shared state management and styling consistency required careful handling (Altexsoft, 2022). However, the study faced limitations, including a limited exploration of tools like Workbox or Turborepo, which could have streamlined development, and the broad scope of micro-frontends, which constrained in-depth analysis due to time limitations.

Additionally, the simplicity of the restaurant website restricted the demonstration of micro-frontends' full potential, as similar performance gains might have been achievable with traditional single-page applications, potentially diminishing the perceived impact of the architecture (Altexsoft, 2022). The analysis of network requests (73% reduction, from 45 to 12) and data transferred (87% reduction, from 2.3MB to 0.3MB), along with a 94% cache hit rate and 100% offline functionality, highlighted the practical benefits of service workers for user experience and cost efficiency (MDN Web Docs, 2024c). These results provide valuable insights into optimizing small-scale applications while acknowledging areas where further refinement is needed.

7.2 Recommendations for Future Development

Transitioning from the study's findings and limitations, several recommendations are proposed for developers pursuing similar projects to enhance the implementation of micro-frontend architectures and service workers in future projects. Developers should prioritize tools like Workbox and Turborepo to streamline the orchestration of remote applications and caching processes within micro-frontend architectures. These tools can simplify complex workflows, improve build efficiency, and ensure robust caching strategies, ultimately reducing development overhead (Altexsoft, 2022). Additionally, focusing on a single topic—such as caching strategies or micro-frontend implementation, would allow for a more in-depth investigation, enabling a comprehensive analysis of specific challenges and solutions. This targeted approach can yield deeper insights and more refined optimizations compared to addressing multiple topics simultaneously.

To fully showcase the scalability and team autonomy benefits of micro-frontends, developers are encouraged to implement more complex applications, such as an online store, where independent teams can work concurrently on different components to accelerate development and improve maintainability (Altexsoft, 2022). Conducting extensive testing across older browsers and a wider range of devices is critical to ensure inclusivity and accessibility for all users. This approach aligns with best practices for creating universally accessible web applications and addresses potential compatibility issues that may arise in diverse user environments (MDN Web Docs, 2025b). Such testing would address potential compatibility issues and broaden the reach of the application, making it more robust for diverse user environments.

For business owners like Mon'adi Ravintola, adopting micro-frontends combined with service workers offers significant practical benefits. The study demonstrated an 87% reduction in data transferred (from 2.3MB to 0.3MB), which can translate to hosting cost savings of up to 87%, alongside improved customer retention driven by faster load times and reliable offline access. These improvements enhance the user experience, making the website more competitive in

the digital landscape. Overall, this project provided valuable practical insights into modern frontend performance optimization techniques, illustrating how the integration of service workers with modular architectures can deliver measurable improvements in real-world applications, paving the way for more efficient, scalable, and user-centric web development.

References

Printed

Martin, K. 2024. The complete developer: Master the full stack with Typescript, React, Next.js, MongoDB, and Docker.

Rappl, Florian & Lothar Schöttner. 2024. The Art of Micro-frontends: Build Highly Scalable, Distributed Web Applications with Multiple Teams. Birmingham: Packt Publishing, Limited. Accessed October 29, 2024. ProQuest Ebook Central.

Vasilijević, V., Kojić, N. and Vugdelija, N. 2020 NEW APPROACH IN QUANTIFYING USER EXPERIENCE IN WEB-ORIENTED APPLICATIONS, in International Scientific Conference ITEMA. Recent Advances in Information Technology, Tourism, Economics, Management and Agriculture, pp. 9–16. doi:10.31410/itema.2020.9.

Vinci, J. R. 2023. Building Micro-frontends with React 18: Develop and Deploy Scalable Applications Using Micro Frontend Strategies. Birmingham: Packt Publishing, Limited. Accessed November 3, 2024. ProQuest Ebook Central.

Electronics

Agarwal, V. & Sastry, N. 2022. Way back then: A Data-driven View of 25+ years of Web Evolution. In Proceedings of the ACM Web Conference 2022 (WWW '22). Accessed October 29, 2024. https://dl-acm-org.nelli.laurea.fi/doi/pdf/10.1145/3485447.3512283

Altexsoft. 2022. A Microservice Approach to Developing Web UIs. Accessed October 28, 2024. https://www.altexsoft.com/blog/micro-frontend/

Atlassian. 2024. Advantages of a monolithic architecture. Accessed November 5, 2024. https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

Berners-Lee, T. 1989. The World Wide Web. Accessed October 21, 2024. https://www.w3.org/People/Berners-Lee/

Chameera Dulanga. 2024. Micro Frontend: Run-Time Vs. Build-Time Integration. Accessed November 12, 2024. https://www.syncfusion.com/blogs/post/micro-frontend-run-time-vs-build-time

Chris, R. 2024. Pattern: Monolithic Architecture. Accessed November 2, 2024. https://microservices.io/patterns/monolithic.html

Cloudflare no date. What is caching? Accessed November 19, 2024. https://www.cloudflare.com/learning/cdn/what-is-caching/

Cloudflare no date. What is time-to-live (TTL)? | TTL definition?. Accessed November 19, 2024. https://www.cloudflare.com/learning/cdn/glossary/time-to-live-ttl/

Codecademy. 2024. What is CRUD? Accessed October 29, 2024. https://www.codecademy.com/article/what-is-crud

Craig, P. no date. Decomposing Monolithic Applications using Separation of Concerns. Accessed November 14, 2024. https://intelligentpathways.com.au/decomposing-monolithic-applications-using-separation-of-concerns/

Google Chrome. 2019a. First Contentful Paint. Accessed April 23, 2025. https://developer.chrome.com/docs/lighthouse/performance/first-contentful-paint

Google Chrome. 2019b. Time to Interactive. Accessed April 23, 2025. https://developer.chrome.com/docs/lighthouse/performance/interactive

Google Chrome. 2021a. Chrome for developers. Accessed November 02, 2024.https://developer.chrome.com/docs/workbox/service-worker-lifecycle

Google Chrome. 2021b. Chrome for developers. Accessed October 15, 2024. https://developer.chrome.com/docs/workbox/caching-strategies-overview

Google Chrome. 2024. Chrome for Developers. Accessed October 15, 2024. https://developer.chrome.com/docs/workbox/service-worker-overview

Google Developer. 2025. Google Search Central. Accessed May 04, 2025.https://developers.google.com/search/docs/fundamentals/seo-starter-guide

PageSpeed Insights. 2024. Accessed December 7, 2024. https://pagespeed.web.dev/

GTmetrix. 2024. Performance Report for BBC News (Generated via GTmetrix Tool). Accessed December 6, 2024. https://gtmetrix.com/

Kapuriya, P. 2022. Web Development Challenges & Solutions You Can't Ignore in 2024. Accessed October 29, 2024. https://www.codzgarage.com/blog/web-development-challenges-solutions

LEGO. 2024. Official web page. Accessed October 28, 2024. https://www.lego.com/en-us

Lighthouse. 2024. Performance Report for Amazon Product Page (Generated via Google Chrome). Accessed October 23, 2024.

MDN Web Docs. 2024a. LocalStorage. Data Persistence. Accessed October 24, 2024. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

MDN Web Docs. 2024b. Making PWAs work. Making PWAs work offline with Service workers. Accessed November 19, 2024. https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Tutorials/js13kGames/Offline_Service_workers

MDN Web Docs. 2024c. Service worker API. Accessed October 29, 2024. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

MDN Web Docs. 2024d. FetchEvent. Accessed April 26, 2025. https://developer.mozilla.org/en-US/docs/Web/API/FetchEvent

MDN Web Docs. 2024e. ServiceWorkerGlobalScope. Accessed April 27, 2025. https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope/install_event

MDN Web Docs. 2024f. Service Workers. Background Synchronization. Accessed November 23, 2024. https://developer.mozilla.org/en-US/docs/Web/API/Background_Synchronization_API

MDN Web Docs. 2024g. Web Performance. The "why" of web performance. Accessed December 4, 2024. https://developer.mozilla.org/en-US/docs/Learn/Performance/why_web_performance

MDN Web Docs. 2025a. Measuring performance. Accessed April 26, 2025. https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Performance/Measuring_performance

MDN Web Docs, Responsive Design. 2025b. Accessed April 26, 2025. https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/CSS_layout/Responsive_Design

Navdeep S. G. 2004. Enterprise Digital Platform, Micro frontend architecture, and best practices. Accessed November 3, 2024. https://www.xenonstack.com/insights/micro-frontend-architecture

Page Load Time. 2025. Page Load Time. Accessed April 24, 2025. https://developer.mozilla.org/en-US/docs/Glossary/Page_load_time

React. No date.a. State. Accessed April 10, 2025. https://react.dev/learn/managing-state

React. No date.b. Lazy Loading. Accessed April 30, 2025. https://react.dev/reference/react/lazy

React. No date.c. Suspense. React. Accessed April 30, 2025. https://react.dev/reference/react/Suspense

Ritesh, Ayush Yadav, Ashutosh Dubey, Deepak Prajapati, Akarsh Srivastava. 2022. Evolution of Web Development Frameworks. International Journal for Research in Applied Science and Engineering Technology, 11(11), 2280. Accessed October 21, 2024. https://www.ijraset.com/best-journal/evolution-of-web-development-frameworks

Saransh Kataria. 2019. How to use a Service Worker to create Progressive Web Applications?. Accessed November 21, 2024. https://www.wisdomgeek.com/development/web-development/service-worker-progressive-web-applications/

Udemy. 2021. Micro-frontends with React: A Complete Developer's Guide. Accessed November 13, 2024. https://www.udemy.com/course/microfrontend-course/?couponCode=24T2MT111524

UDN Web Docs. 2020. Service Workers: Using Service Workers. Accessed November 23, 2024. https://udn.realityripple.com/docs/Web/API/Service_Worker_API/Using_Service_Workers

Vercel. No date. Vercel platform. Accessed May 1, 2025. from https://vercel.com/

Vishal Sharma. 2023. You Don't Need Another Library to Compose Micro-frontends at Run Time. Accessed November 12, 2024. https://blog.bitsrc.io/you-dont-need-another-library-to-compose-micro-frontends-at-run-time-e803077ade67

Vivek Shukla. 2023. A Comprehensive Guide to Micro Frontend Architecture. Accessed December 5, 2024. https://medium.com/appfoster/a-comprehensive-guide-to-micro-frontend-architecture-cc0e31e0c053-run-time-e803077ade67

VsCode. No date. Accessed November 15, 2024. https://code.visualstudio.com/download

Webpack, 2021. ModuleFederationPlugin. Accessed November 13, 2024. https://webpack.js.org/plugins/module-federation-plugin/

Web Vitals . 2024. Accessed April 26, 2025. https://web.dev/articles/vitals

DEV Community. 2023. Web Worker, Service Worker, and Worklets: A Comprehensive Guide. Accessed April 26, 2025. https://dev.to/bharat5604/web-worker-service-worker-and-worklets-a-comprehensive-guide-1f64

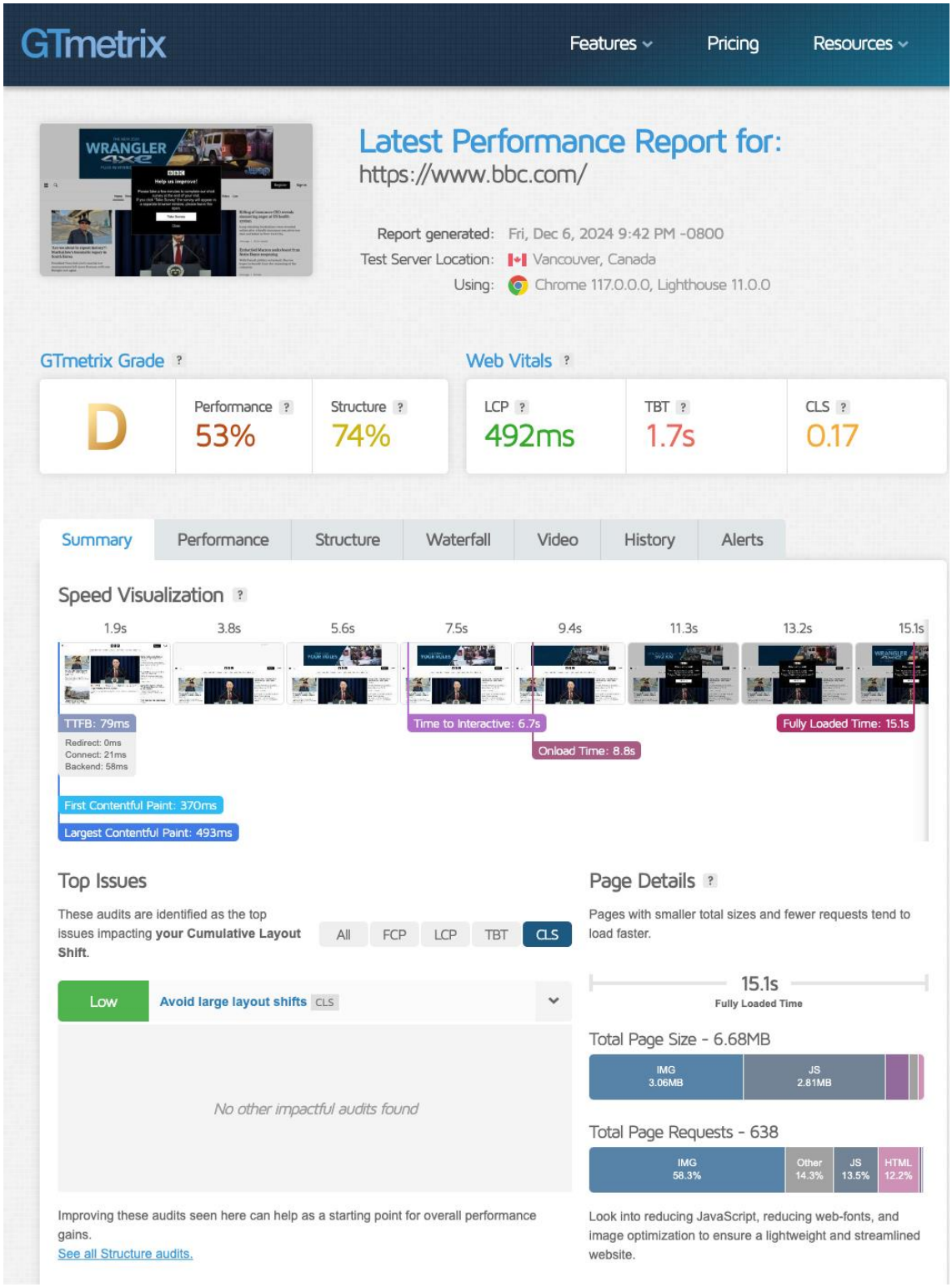
Youtube. Service Workers in JavaScript. 2022. Accessed November 29, 2024. https://www.youtube.com/watch?v=1usuYqZMT7Q

Figures

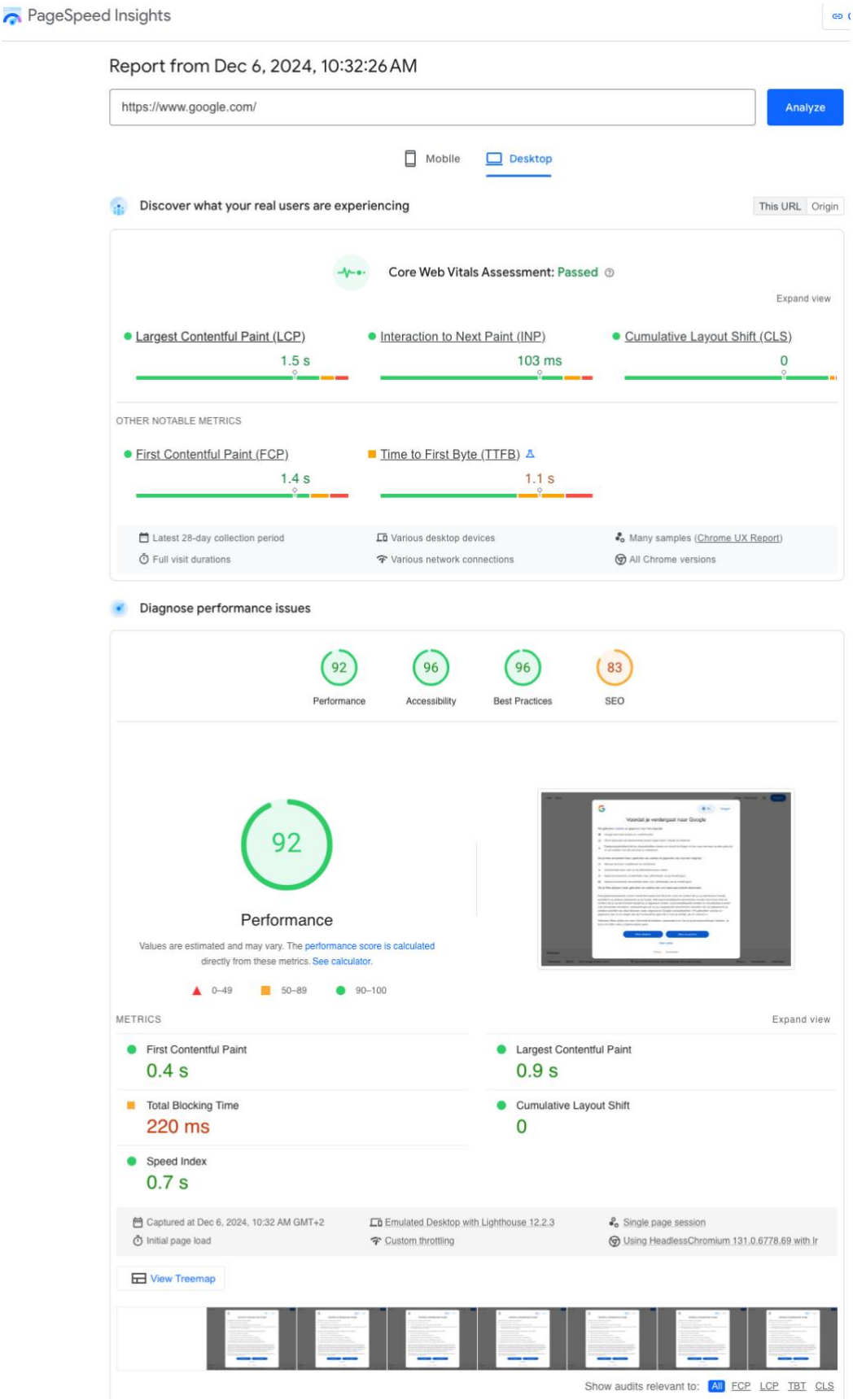
Tables

Appendices

Appendix 1: BBC Website Performance Complete Matrix



Appendix 2: Google Search Performance Complete Matrix



Appendix 3: Amazon Product Page Performance Complete Matrix