

UNIVERSITETI PUBLIK “KADRI ZEKA” GJILAN
FAKULTETI I SHKENCAVE KOMPJUTERIKE
INXHINIERI SOFTUERIKE DHE WEB DIZAJN



PUNIM DIPLOME

Titulli i temës: **Optimizimi dhe arkitekturimi i web aplikacioneve
React të shkallës së lartë**

Mentori:

Prof.Dr. Basri Ahmedi

Kandidatja:

Era Ibrahim

Gjilan, Tetor 2026

UNIVERSITETI PUBLIK “KADRI ZEKA” GJILAN
FAKULTETI I SHKENCAVE KOMPJUTERIKE
INXHINIERI SOFTUERIKE DHE WEB DIZAJN



PUNIM DIPLOME

**Titulli i temës: Optimizimi dhe arkitekturimi i web aplikacioneve
React të shkallës së lartë**

Lënda:	Zhvillimi i web aplikacioneve
Titulli i punimit:	Optimizimi dhe arkitekturimi i web aplikacioneve React të shkallës së lartë
Emri dhe Mbiemri:	Era Ibrahim
Statusi i studentit:	I rregullt
Numri i amzës:	22071051
Niveli i studimeve:	Bachelor
Drejtimi:	Fakulteti i Shkencave Kompjuterike
Programi i studimit:	Inxhinieri Softuerike dhe Web Dizajn
Mentori i punimit:	Prof.Dr. Basri Ahmed

Gjilan, Tetor 2026

Tabela e Përmbajtjes

Lista e Figurave	6
Lista e Shkurtesave	7
Deklarata	8
Falënderime	9
Abstrakti	10
Abstract	11
1. Hyrje	12
2. SPA-të tradicionale dhe kufizimet e tyre	13
2.1. Çka jane SPA-të	13
2.2. React dhe CSR (Client Side Rendering)	14
2.3. Problemet e qasjes tradicionale	14
3. Analiza e Metrikave të Performancës (Web Vitals)	16
3.1. Web Vitals dhe Core Web Vitals	16
3.2. Sfidat e metrikave në arkitekturat React (CSR)	17
3.2.1. LCP dhe vonesa e zbulimit të burimeve	17
3.2.2. Nga FID te INP: Matja e ndërveprimit të vazhdueshëm	19
3.2.3. CLS dhe dinamizmi asinkron	19
3.3. Pasqyra e vlerave standarde	19
3.4. Ndërlidhja mes Arkitekturës dhe Metrikave	19
4. Server-Side Rendering (SSR) — Renderimi në anën e serverit	21
4.1. Çfarë është SSR	21
4.2. Si funksionon SSR me React dhe Next.js	23
4.3. Ndikimi i SSR në Web Vitals	23
4.4. Kufizimet e SSR	24
5. Static Site Generation (SSG) – Gjenerimi Statik i Faqeve	25
5.1. Çfarë është SSG	25
5.2. Si funksionon SSG me Next.js	26
5.3. ISR: Incremental Static Regeneration	26
5.4. SSG dhe Web Vitals	27
5.5. Krahasimi SSR vs SSG vs CSR	27
6. Streaming dhe React Suspense	28
6.1. Kufizimi i SSR klasik: Bllokimi "All-or-Nothing"	28
6.2. Streaming SSR me React 18	28

6.3.	Selective Hydration	29
6.4.	React Server Components (RSC)	29
6.5.	Ndikimi i Streaming dhe RSC në Web Vitals	29
7.	Modularizimi dhe Optimizimi i JS Bundle	30
7.1.	Problemi i Bundle-it të Madh	30
7.2.	Code Splitting	30
7.3.	Lazy Loading i Komponentëve	31
7.4.	Tree Shaking	31
7.5.	Optimizimi i Varësive të Palës së Tretë	31
7.6.	Strategjia e Optimizimit të Imazheve	32
8.	Studimi i rastit – Migrimi nga CSR ne SSR/SSG	32
8.1.	Përshkrimi i Skenarit	32
8.2.	Matjet Fillestare (Para Migrimit)	32
8.3.	Strategjia e Migrimit	33
8.4.	Matjet Pas Migrimit	34
8.5.	Vërejtje metodologjike	36
9.	Përfundimi dhe rekomandime	38
9.1.	Përfundimi	38
9.2.	Rekomandime Praktike	38
9.3.	Perspektiva dhe punë e ardhshme	39
	Referencat	40

Lista e Figurave

Figure 1 SPA - Arkitektura	13
Figure 2 Core Web Vitals	16
Figure 3 LCP - Vonesa e zbulimit të burimeve	18
Figure 4 SSR – cikli jetësorë i kërkesës në Next.js	22
Figure 5 SSG - Procesi i SSG në Fazën e Ndërtimit dhe të Shërbimit	26
Figure 6 CSR – Metrikat live nga Chrome DevTools (para migrimit)	35
Figure 7 Next.js – Metrikat live nga Chrome DevTools (pas migrimit)	36

Lista e Shkurtesave

SPA – Aplikacion në një faqe (Single Page Application)
MPA – Aplikacion me shumë faqe (Multiple page Application)
XSS – Skriptim ndër faqesh (Cross Site Scripting)
UX – Eksperienca e përdoruesit (User Experience)
UI – Ndërfaqja e përdoruesit (User Interface)
SEO – Optimizimi i makinave të kërkimit (Search Engine Optimization)
TTFB – Koha deri në bitin e parë (Time to First Byte)
FCP – Pikturimi i parë i përmbajtjes (First Contentful Paint)
LCP – Pikturimi më i madh i përmbajtjes (Largest Contentful Paint)
FID – Vonesa e parë e ndërveprimit (First Input Delay)
INP – Ndërveprimi deri në pikturimin e rradhës (Interaction to Next Paint)
CLS – Zhvendosja kumulative e paraqitjes (Cumulative Layout Shift)
CSR – Renderimi në anën e klientit (Client-Side Rendering)
SSR – Renderimi në anën e serverit (Server-Side Rendering)
SSG – Gjenerimi i faqes statike (Static Site Generation)
CDN – Rrjet i Shpërndarjes së Përmbajtjes (Content Delivery Network)
ORM – Hartimi Objekt-Relacional (Object-Relational Mapping)
RUM – Monitorim real i përdoruesve (Real User Monitoring)
RTT – Tekst në kohë reale (Real Time Text)

Deklarata

Unë Era Ibrahim, studente në Universitetin “Kadri Zeka” në Gjiilan, në Departamentin e Shkencave Kompjuterike, Drejtimi SEW (Software Engineering and Web Design – Inxhinieri Softuerike dhe Dizajn Web me numër të ID-së: 22071051, deklaroj me përgjegjësi të plotë se ky punim është rezultati i punës sime personale, të zhvilluar gjatë viteve të studimeve dhe përvojave praktike në projekte të ndryshme. Literatura dhe burimet e përdorura janë referuar me kujdes, në përputhje me normat dhe standartet akademike. Ky punim nuk është paraqitur më parë në ndonjë institucion tjetër, as nuk është publikuar apo përdorur në ndonjë mënyrë tjetër.

Nënshkrimi

Falënderime

Një falënderim i veçantë shkon për familjen time, për mbështetjen, durimin dhe besimin e pakushtëzuar që kanë treguar gjatë gjithë rrugëtimit tim akademik — pa ta do të ishte shumë më vështirë të arrija deri këtu.

Falënderoj Profesorat/et me udhëheqjen e të cilëve ka qenë nxënia e mësimave dhe përgaditja akademike një përvojë e mrekullueshme.

Në fund, falënderoj shokët dhe shoqet e studimeve, për bashkëpunimin, inkurajimin dhe momentet e ndara gjatë këtyre viteve.

Abstrakti

Ky punim trajton optimizimin dhe arkitekturimin e aplikacioneve ueb të ndërtuara me React në shkallë të lartë. Qëllimi është të lidhen në mënyrë të qartë modelet e renderimit (renderimi në anën e klientit, në server dhe gjenerimi statik), matjet e përvojës së përdoruesit përmes Web Vitals dhe organizimi i kodit në komponentë dhe module. Theksohet se suksesi afatgjatë i një faqeje në ueb lidhet me cilësinë e përvojës së përdoruesit dhe se ekziston një grup metrikash të unifikuara që e përfaqësojnë këtë cilësi (Google, 2020). Në pjesën analitike përshkruhen kufizimet e aplikacioneve në një faqe dhe të renderimit në klient, më pas metrikat kryesore të ngarkimit, ndërveprimit dhe qëndrueshmërisë vizuale, si dhe roli i arkitekturës në uljen e madhësisë së paketave JavaScript dhe në përmirësimin e ngarkimit fillestar (GeeksforGeeks, 2026; Google, 2020; Meta, n.d.; Vercel, 2026). Punimi përfundon me rekomandime praktike dhe një kornizë për studim rasti me matje para dhe pas ndryshimeve arkitekturore.

Fjalë kyçe: React, Web Vitals, SPA, optimizim, arkitekturë, renderim

Abstract

This thesis addresses optimization and architecture of large-scale React web applications. It connects rendering models (client-side, server-side, and static generation), user-experience measurement through Web Vitals, and code organization. The work argues that durable site quality depends on user experience and that unified metrics help sites focus on what matters most (Google, 2020). It summarizes limitations of single-page apps and client-side rendering, then discusses core loading, interactivity, and visual-stability metrics, and how architecture affects JavaScript bundle size and first paint (GeeksforGeeks, 2026; Google, 2020; Meta, n.d.; Vercel, 2026). The thesis closes with practical recommendations and a case-study outline with before-and-after measurements.

Keywords: React, Web Vitals, SPA, optimization, architecture, rendering.

1. Hyrje

Zhvillimi i aplikacioneve ueb sot shpesh përdor librari dhe korniza që organizojnë ndërfaqen në komponentë dhe që lejojnë përditësime dinamike pa rifreskim të plotë të faqes. Një aplikacion në një faqe ngarkon përmbajtje sipas veprimeve të përdoruesit dhe përdor JavaScript për të përditësuar vetëm pjesët e nevojshme të ekranit, duke reduktuar rifreskimet e plota dhe duke kërkuar më pak të dhëna nga serveri kur është e mundur (GeeksforGeeks, 2026). React është zgjedhur gjerësisht për këtë lloj aplikacioni sepse modeli i komponentëve dhe cikli i renderimit lehtëson menaxhimin e ndërfaqes ndërsa aplikacioni rritet (Meta, n.d.).

Në aplikacione të mëdha, nevoja për performancë të matshme, për përvojë të qëndrueshme për përdoruesit me pajisje dhe rrjet të ndryshëm dhe për organizim kodi që mbetet i mirëmbajtshëm e bën të domosdoshme lidhjen mes arkitekturës dhe optimizimit. Iniciativa Web Vitals ofron udhëzime të përbashkëta për sinjalet e cilësisë që lidhen me përvojën e përdoruesit dhe thjeshton peizazhin e shumë metrikave duke theksuar ato që quhen Core Web Vitals (Google, 2020).

Objektivat e punimit janë: (1) të përshkruhen kufizimet e qasjes klasike SPA/CSR në kontekstin e shkallës së lartë; (2) të sistematizohen metrikat kryesore të performancës së përvojës dhe mënyra se si lidhen me ngarkimin fillestar, ndërveprimin dhe stabilitetin vizual (Google, 2020; Google, 2025); (3) të argumentohet se optimizimet lokale në nivel komponenti nuk zëvendësojnë një ndarje të mirë përgjegjësishe dhe modulesh kur problemi është struktural (Meta, n.d.); (4) të paraqitet si zgjidhje e lidhur natyrshëm reduktimi i JavaScript-it të dërguar në shfletues dhe përdorimi i renderimit në server ose gjenerimit statik kur përmbajtja dhe kërkesat e projektit e lejojnë (GeeksforGeeks, 2026; Vercel, 2026).

Metodologjia është kryesisht analitike dhe përmbledhëse, me mbështetje në dokumentacion zyrtar dhe në artikuj të specializuar ueb-i; pjesa e studimit të rastit mbetet për implementim konkret dhe matje (Google, 2020).

Struktura e punimit vazhdon me SPA-të dhe kufizimet e tyre, më pas me analizën e metrikave dhe lidhjen arkitekturë–performancë, me modelet SSR/SSG dhe modularizimin, dhe përfundon me përfundim dhe rekomandime.

2. SPA-të tradicionale dhe kufizimet e tyre

2.1. Çka janë SPA-të

Siç paraqitet në figurën e mëposhtme, në një arkitekturë Single-Page Application (SPA), shfletuesi ngarkon fillimisht një faqe të vetme dhe më pas komunikon vazhdimisht me serverin përmes API kërkesave, ndërsa përditësimet e ndërfaqes realizohen nga JavaScript në anën e klientit. Kjo qasje ndryshon ndjeshëm nga MPA-të (Multi-Page Applications), të cilat bëjnë renderimin e plotë të faqes në server për çdo ndërveprim të përdoruesit. Në një SPA, ekziston vetëm një dokument HTML fillestar, i cili përditësohet në mënyrë dinamike pa kërkuar rifreskimin e plotë të faqes.

SPA përdor JavaScript për të përditësuar vetëm pjesët e prekura nga ndryshimi. Si rrjedhojë, SPA-të ofrojnë një përvojë përdoruesi (UX) të ngjashme me atë të aplikacioneve desktop, me ndërveprim të lartë dhe shpejtësi në servimin e të dhënave nga serveri. Në modelin e paraqitur në figurë, logjika e aplikacionit ekzekutohet kryesisht në shfletues, ndërsa serveri shërben kryesisht si burim i të dhënave përmes API.

Megjithëse SPA-të sollën një revolucion në aspektin e UX, ato paraqesin sfida dhe kufizime të qarta kur bëhet fjalë për aplikacione të shkallës së lartë, veçanërisht në optimizimin për motorët e kërkimit (SEO) dhe performancën fillestare (GeeksforGeeks, 2026).

Burimi: <https://www.altexsoft.com/blog/web-application-architecture-how-the-web-works>

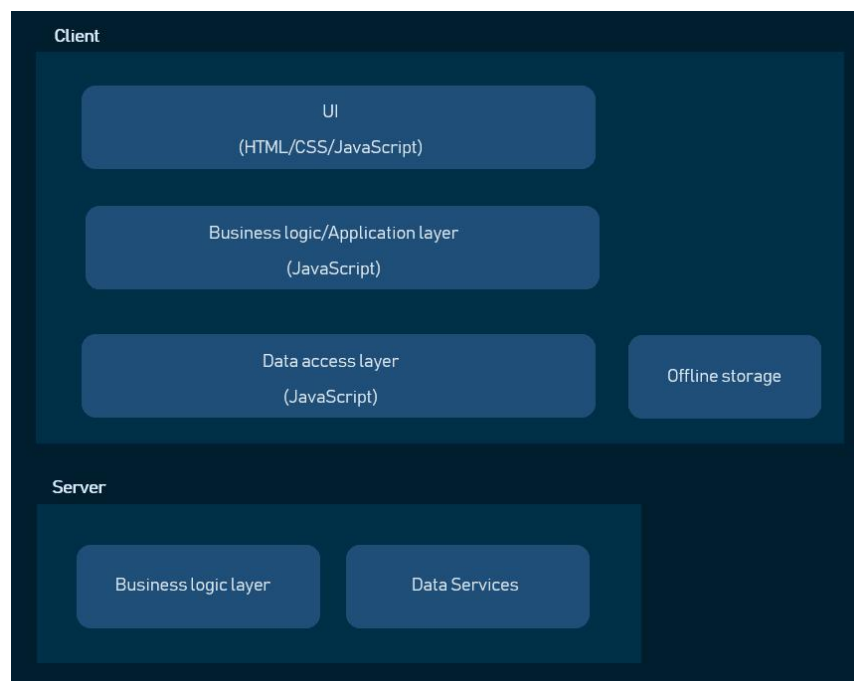


Figure 1 SPA - Arkitektura

2.2. React dhe CSR (Client Side Rendering)

Arsyeja pse libraria React përdoret përgjithësisht për SPA qëndron tek arkitektura e saj e bazuar në komponente, e cila tregon përshtatshmëri të lartë me SPA-të në kontekstin e izolimit të kodit dhe përditësimeve specifike, duke shmangur renderimet e tepërta. Përveç kësaj, DOM-i virtual i React-it, i cili optimizon procesin e përditësimit të UI, luan një rol kyç në përmbushjen e qëllimit të SPA-ve. Gjithashtu, ekosistemi i gjerë i React-it, së bashku me fleksibilitetin e tij, shërbejnë si avantazhe shtesë në leverdinë që kjo teknologji ofron.

Në mënyrën klasike, React përdorte **CSR** (Client-Side Rendering), ku procesi i renderimit ndodh plotësisht në klient (shfletues). Ky proces zakonisht ndjek këta hapa:

1. Kod HTML minimal

Përdoret një strukturë boshe, zakonisht: `<div id="root"></div>`

2. Ngarkimi i "JavaScript bundle"

Ngarkohet një skedar, ose disa skedarë JavaScript, të cilët përmbajnë të gjithë logjikën e nevojshme për aplikacionin, e njohur ndryshe si **source code**.

3. Renderimi në shfletues (browser)

Vetëm pasi JavaScript-i të ekzekutohet, React fillon të ndërtojë ndërfaqen dhe të shfaqë përmbajtjen për përdoruesin.

Rrjedhimisht, në këtë model, shfletuesi (client) merr përsipër të gjithë peshën e ekzekutimit të logjikës së aplikacionit, manipulimin e DOM-it dhe renderimin vizual të ndërfaqes, duke e shndërruar pajisjen e përdoruesit në motorin kryesor të përpunimit të të dhënave.

2.3. Problemet e qasjes tradicionale

Megjithëse *Client-Side Rendering* është i thjeshtë për t'u implementuar, ai paraqet probleme serioze kur aplikohet në aplikacione komplekse dhe të shkallës së gjerë. Sfidat kryesore përfshijnë:

- **Faqe e zbrazët gjatë ngarkimit fillestar**

Derisa JavaScript-i të shkarkohet dhe të ekzekutohet plotësisht, përdoruesi shpesh përballet me një ekran të zbrazët, gjë që krijon një përvojë fillestare jo të mirë.

- **Kohë e lartë e renderimit fillestar**

Paketat e mëdha të JavaScript-it (*bundles*) rrisin ndjeshëm kohën e nevojshme deri në shfaqjen e përmbajtjes së parë, veçanërisht në pajisje me resurse të kufizuara ose në lidhje të dobëta interneti.

- **UX dhe problematika SEO**

Motorët e kërkimit (SEO) dhe përdoruesit nuk pranojnë menjëherë përmbajtje të renderuar, e cila ndikon negativisht në indeksim, në renditjen e faqes dhe në perceptimin e përgjithshëm të performancës .

- **Varësi e lartë nga JavaScript-i**

Çdo funksionalitet i aplikacionit varet tërësisht nga ekzekutimi korrekt i kodit JavaScript. Kjo e bën aplikacionin më të ndjeshëm ndaj gabimeve gjatë ekzekutimit (runtime errors), dështimeve të rrjetit apo kufizimeve teknike të shfletuesit. Në këtë model, një gabim i vetëm kritik në kod mund të bëjë që të gjithë komponentët e faqes të dështojnë (crash), duke e bërë aplikacionin plotësisht të papërdorshëm. Ndonëse këto raste mund të menaxhohen përmes mekanizmave si Error Boundaries në React, ky nivel varësie rrit ndjeshëm rrezikun dhe kërkon kujdes të shtuar gjatë zhvillimit.

Si pasojë, kjo qasje tradicionale e SPA-ve, megjithëse funksionale për aplikacione të vogla dhe të mesme, ekspozon qartë dobësitë e saj në kontekstin e **performancës, shkallëzueshmërisë dhe optimizimit**. (Osmani, a.d.)

3. Analiza e Metrikave të Performancës (Web Vitals)

3.1. Web Vitals dhe Core Web Vitals

Web Vitals është një iniciativë që përmbledh udhëzime për sinjalet e cilësisë thelbësore për një përvojë të mirë në ueb dhe synon të thjeshtësojë peizazhin e metrikave duke theksuar ato që quhen Core Web Vitals. Grupi aktual i Core Web Vitals mbulon tri dimensione: ngarkimi, ndërveprimi dhe qëndrueshmëria vizuale; për secilin ekzistojnë pragje të rekomanduara dhe matja e përshtatshmërisë shpesh merret në përqindjen e 75-të të ngarkimeve (Google, 2020).

Burimi: <https://dmidigitalmarketing.com/the-ultimate-guide-to-core-web-vitals-metrics-interaction-to-next-paint-inp/>

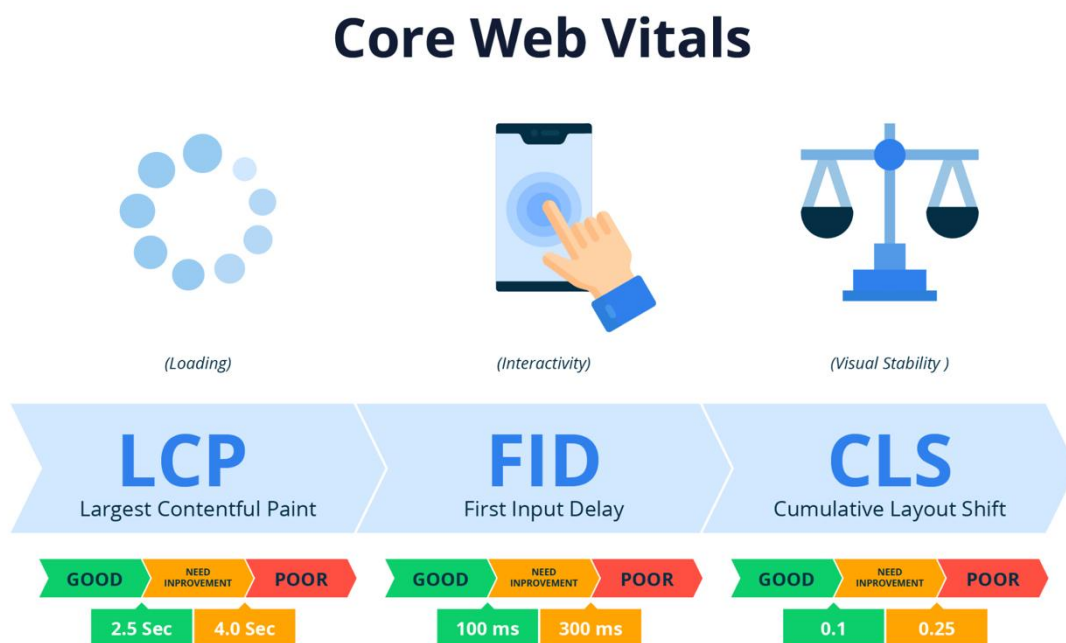


Figure 2 Core Web Vitals

Metrika Largest Contentful Paint (LCP) lidhet me performancën e **ngarkimit**; për një përvojë të mirë rekomandohet që LCP të arrihet brenda 2.5 sekondave nga fillimi i ngarkimit të faqes. LCP reflekton kohën kur elementi më i madh përmbajtës në pamje (p.sh. imazh, bllok teksti ose video në viewport) është renderuar krahasuar me momentin kur përdoruesi filloi navigimin; në praktikë

në këtë kohë përfshihen edhe vonesat e lidhura me TTFB (Time To First Byte) dhe hapat para renderimit të përmbajtjes kryesore (Google, 2025).

Interaction to Next Paint (INP) mat **ndërveprimin**; rekomandohet prag prej 200 ms ose më pak për përvojë të mirë (Google, 2025). Cumulative Layout Shift (CLS) mat **stabilitetin vizual**; rekomandohet të mbetet nën ose në 0.1 për përvojë të mirë (Google, 2023).

Metrika të tjera si Time to First Byte dhe First Contentful Paint ndihmojnë të diagnostohen probleme që ndikojnë te ngarkimi dhe te LCP, p.sh. përgjigje e ngadaltë e serverit ose burime që bllokojnë renderimin (Google, 2025).

3.2. Sfidat e metrikave në arkitekturat React (CSR)

Për të kuptuar ndikimin e arkitekturës në performancë, duhet të analizojmë se si këto metrika sillen në mjedisin e një aplikacioni tradicional React ku renderimi ndodh në klient.

3.2.1. LCP dhe vonesa e zbulimit të burimeve

Në aplikacionet React (CSR), LCP shpesh vuan nga ajo që njihet si "vonesa e zbulimit". Duke qenë se HTML-i fillestar është bosh, shfletuesi nuk mund të identifikojë imazhin kryesor apo bllokun e tekstit LCP derisa të shkarkohet dhe ekzekutohet i gjithë "bundle-i" JavaScript. Kjo krijon një zinxhir varësish si në vijim:

1. Marrja e HTML-it (TTFB).
2. Shkarkimi i JS.
3. Ekzekutimi i React.
4. Kërkesa API për të dhënat.
5. **Renderimi i LCP.** Ky proces shpesh e shtyn LCP-në përtej pragut kritik prej 2.5 sekondash, duke penalizuar aplikacionin në aspektin e performancës së perceptuar nga përdoruesit (Google, 2025).

Burimi: Fotografi e autorit

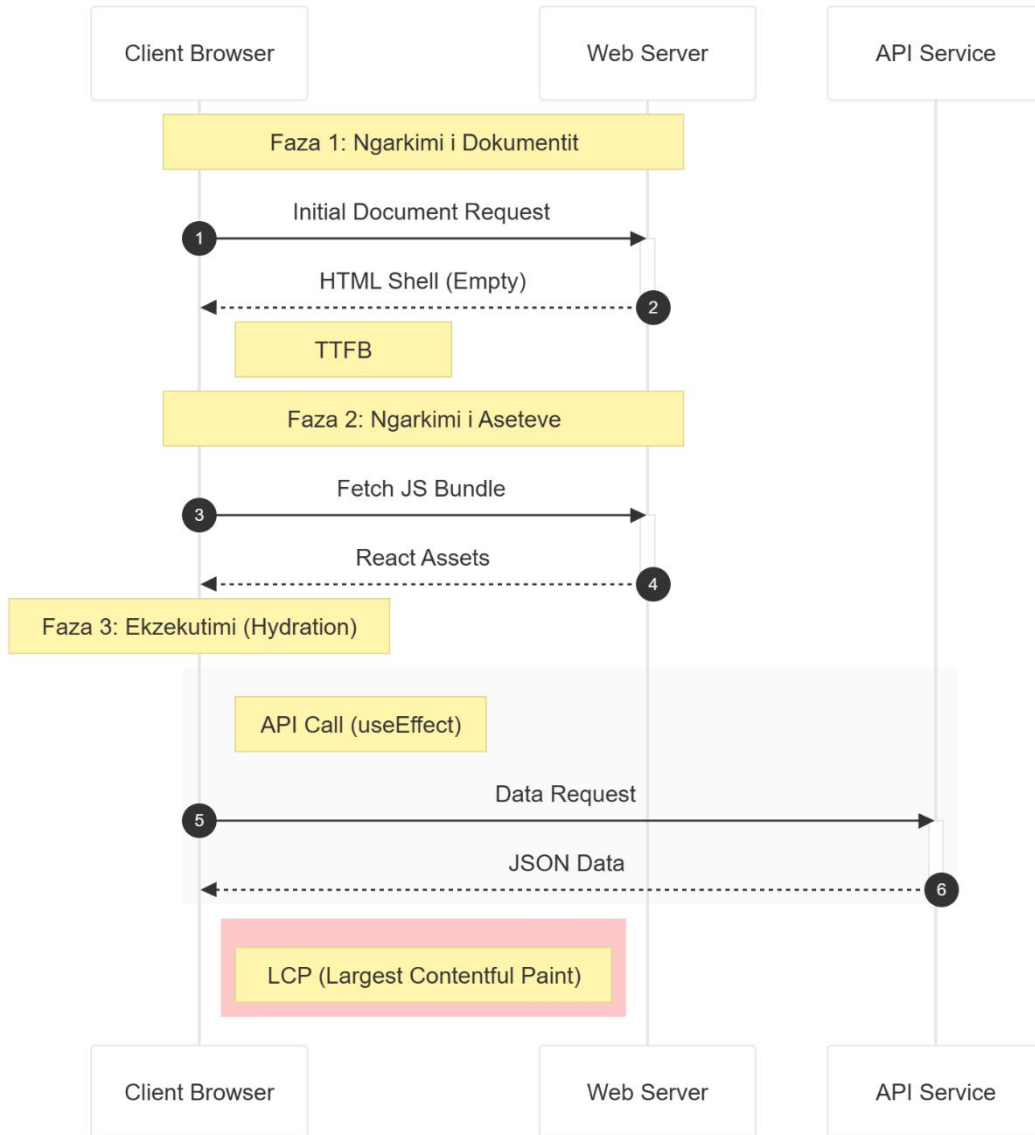


Figure 3 LCP - Vonesa e zbulimit të burimeve

3.2.2. Nga FID te INP: Matja e ndërveprimit të vazhdueshëm

Ndonëse FID (*First Input Delay*) ishte metrika standarde, ajo maste vetëm vonesën e parë pas ngarkimit. Për aplikacionet e shkallës së lartë, ku përdoruesi navigon gjatë pa rifreskuar faqen, **INP (Interaction to Next Paint)** ofron një pasqyrë më reale dhe të rëndësishme.

Në React, sfida kryesore për INP mbetet bllokimi i fijes kryesore (*Main Thread*). Gjatë procesit të *Hydration* (procesi i bashkëngjijtes së dëgjuesve të eventeve dhe logjikës së gjendjes në HTML statike SSR ose SSG) ose gjatë ri-renderimeve të rënda të komponentëve, procesori është i zënë duke ekzekutuar kodin JavaScript. Nëse një përdorues ndërvepron (p.sh. klikon një menu) gjatë këtij momenti, shfletuesi nuk mund të përgjigjet vizualisht derisa procesi i React të përfundojë, duke shkaktuar një vlerë të lartë të INP-së.

3.2.3. CLS dhe dinamizmi asinkron

Stabiliteti vizual (CLS) në React preket kryesisht nga ngarkimi asinkron i komponentëve. Shfaqja e papritur e një elementi pasi të dhënat janë marrë nga serveri mund të zhvendosë elementet ekzistuese poshtë. Në aplikacione të shkallës së lartë, kjo kërkon teknika arkitekturore si "Skeleton Screens" ose rezervimi i hapësirës (min-height) për të parandaluar kërcimin e ndërfaqes.

3.3. Pasqyra e vlerave standarde

Për të ofruar një kornizë vlerësimi gjatë studimit të rastit në këtë punim, do t'u referohemi pragjeve zyrtare të Google për Core Web Vitals:

Metrika	<i>Mirë (Good)</i>	<i>Nevojitet Përmirësim</i>	<i>Dobët (Poor)</i>
LCP (Loading)	≤ 2.5 sekonda	2.5s - 4.0s	> 4.0 sekonda
INP (Interactivity)	≤ 200 ms	200ms - 500ms	> 500 ms
CLS (Visual Stability)	≤ 0.1	0.1 - 0.25	> 0.25

3.4. Ndërlidhja mes Arkitekturës dhe Metrikave

Si përfundim i kësaj analize, vërehet se problemet e performancës në aplikacionet React të shkallës së lartë nuk janë thjesht pasojë e kodit të shkruar keq, por rrjedhin nga vetë natyra e arkitekturës **CSR**. Varësia e tepërt nga ekzekutimi i JavaScript-it në klient krijon një "ngushticë" (**bottleneck**) që ndikon negativisht në të gjitha metrikat *Core Web Vitals*.

Për të adresuar këto kufizime, optimizimi i thjeshtë i komponentëve nuk mjafton. Nevojitet një **rishikim i arkitekturës**, duke kaluar drejt modeleve që shpërndajnë peshën e renderimit mes serverit dhe klientit. Kapitujt në vijim do të trajtojnë strategjitë arkitekturore si **Server-Side Rendering (SSR)**, **Static Site Generation (SSG)** dhe **Streaming**, si dhe rolin e tyre në eliminimin e vonesave të zbulimit të burimeve dhe përmirësimin e ndërveprueshmërisë.

4. Server-Side Rendering (SSR) — Renderimi në anën e serverit

4.1. Çfarë është SSR

Siç ilustruhet në figurën e mëposhtme, në modelin Server-Side Rendering (SSR) procesi fillon kur shfletuesi dërgon një kërkesë fillestare te serveri. Serveri ekzekuton logjikën e aplikacionit, merr të dhënat e nevojshme nga API ose burime të tjera të të dhënave dhe më pas renderon komponentët React në HTML përpara se përgjigja të dërgohet te klienti. Si rezultat, përdoruesi merr menjëherë një faqe të plotë me përmbajtje të gatshme për lexim, pa pritur shkarkimin dhe ekzekutimin e JavaScript-it për të parë informacionin kryesor.

Server-Side Rendering (SSR) është modeli i renderimit ku serveri gjeneron HTML-in e plotë të faqes për çdo kërkesë të klientit, përpara se ky HTML të dërgohet në shfletues. Në ndryshim nga CSR, ku shfletuesi merr një dokument HTML pothuajse bosh dhe më pas ndërton ndërfaqen nëpërmjet JavaScript-it, SSR i ofron shfletuesit një faqe **tashmë të renderuar** dhe të lexueshme menjëherë pas marrjes së përgjigjes nga serveri. Siç paraqitet në figurë, elementi kryesor i përmbajtjes (LCP) mund të shfaqet menjëherë pas marrjes së HTML-it, ndërsa JavaScript-i shkarkohet dhe ekzekutohet më vonë gjatë procesit të hydration-it për ta bërë faqen plotësisht interaktive.

Ky model ekziston që nga fillimet e internetit, ku teknologjitë si PHP, Ruby on Rails dhe ASP.NET gjeneronin faqe HTML dinamike në server. Megjithatë, rindërtimi i SSR-it brenda ekosistemit modern React — me korniza si Next.js — sjell avantazhet e renderimit në server duke ruajtur njëkohësisht fleksibilitetin e komponentëve React dhe interaktivitetin e klientit pas ngarkimit fillestar (Vercel, 2026).

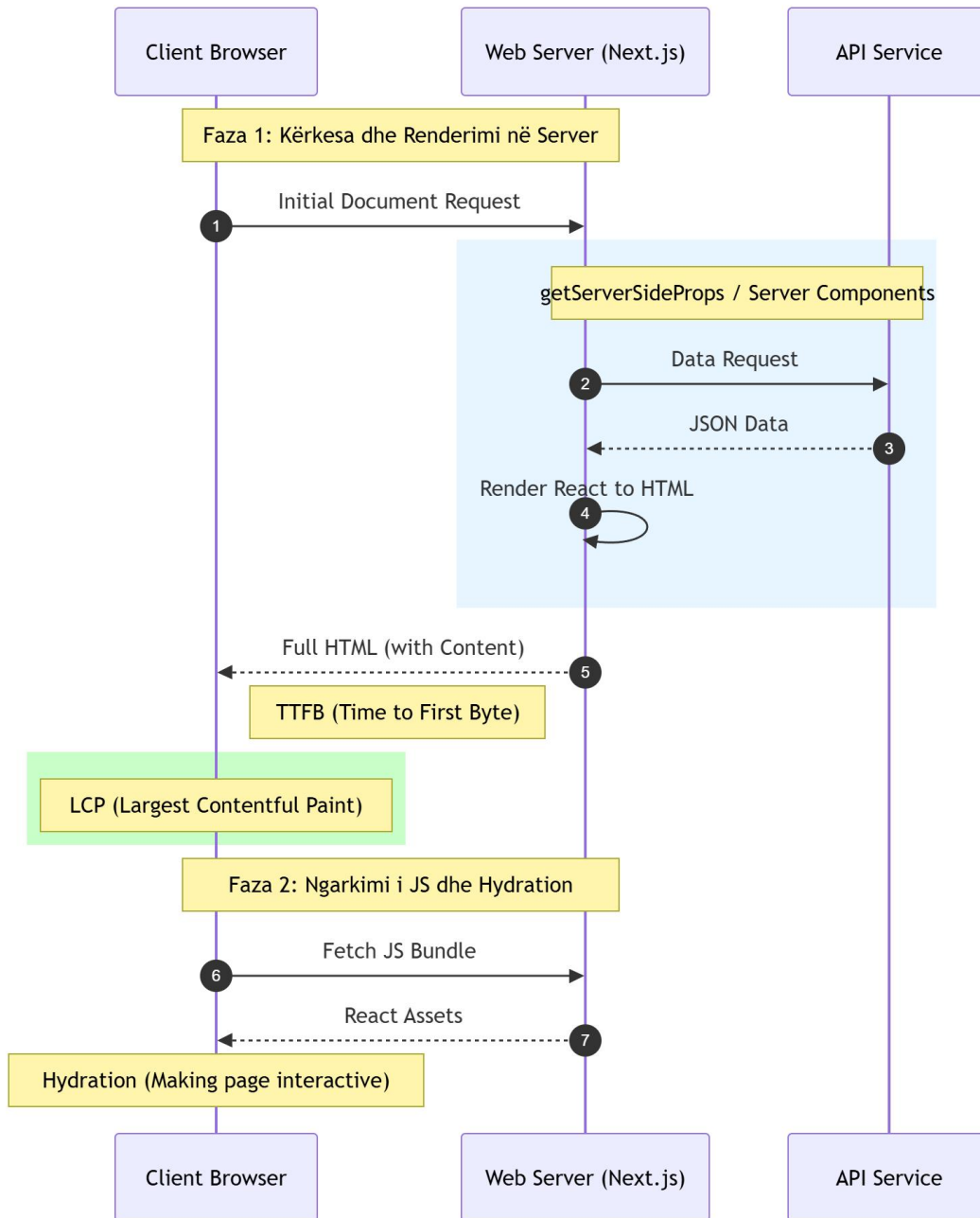


Figure 4 SSR – cikli jetësorë i kërkesës në Next.js

4.2. Si funksionon SSR me React dhe Next.js

Procesi i renderimit SSR ndjek hapat e mëposhtëm:

1. **Kërkesa e klientit** — Përdoruesi kërkon një URL; kërkesa mbërrin në server.
2. **Ekzekutimi i logjikës së serverit** — Serveri ekzekuton kodin React (getServerSideProps në Next.js Pages Router, ose direkt në Server Components me App Router) dhe merr të dhënat e nevojshme nga API ose databaza.
3. **Gjenerimi i HTML-it** — React rendëron komponentët në anën e serverit dhe prodhon një varg HTML të plotë.
4. **Dërgimi te klienti** — HTML-i i plotë dërgohet si përgjigje HTTP; shfletuesi e shfaq menjëherë pa pritur JavaScript-in.
5. **Hydration** — Pasi JavaScript-i të shkarkohet dhe ekzekutohet, React "I bashkëngjet" (hydrate) ndërfaqen ekzistuese HTML me logjikën interaktive – JS bundle, duke e bërë faqen plotësisht funksionale.

Ky model redukton ndjeshëm kohën deri në shfaqjen e përmbajtjes fillestare, pasi shfletuesi nuk pret ekzekutimin e JavaScript-it për të parë përmbajtje (Vercel, n.d.).

4.3. Ndikimi i SSR në Web Vitals

SSR ka ndikim të drejtpërdrejtë dhe të matshëm në metrikat Core Web Vitals:

Largest Contentful Paint (LCP): Për shkak se HTML-i i plotë mbërrin që me përgjigjen e parë të serverit, shfletuesi mund të zbulojë dhe të renderojë elementin LCP (imazhin kryesor ose bllokun e tekstit) shumë herë. Zinxhiri i varësive që ndikon negativisht te LCP në CSR (TTFB → JS Download → JS Execute → API Fetch → Render LCP) shkurtohet ndjeshëm: HTML-i tashmë përmban të dhënat dhe elementet vizuale. Studime të performancës tregojnë se kalimi nga CSR në SSR mund të ulë LCP-në me 30–60% për faqet me përmbajtje të pasur (Google, 2025).

Time to First Byte (TTFB): TTFB rritet pak me SSR krahasuar me shërbimin e një fajlli statik HTML, sepse serveri duhet të ekzekutojë logjikën dhe të marrë të dhënat. Megjithatë, kjo kosto është shumë më e vogël sesa vonesa totale e CSR, dhe mund të optimizohet me caching.

Cumulative Layout Shift (CLS): Meqë HTML-i fillestar tashmë përmban strukturën e plotë të faqes, shmangët zhvendosja tipike e elementeve që ndodh kur komponentët CSR shfaqen me vonesë pas ngarkimit asinkron (Google, 2023).

4.4. Kufizimet e SSR

SSR nuk është pa kosto. Eksistojnë disa kufizime të rëndësishme duhen marrë parasysh:

1. **Ngarkesa e serverit:** Çdo kërkesë provokon ekzekutimin e kodit React dhe mundësisht kërkon akses në bazë të dhënash ose thirrje API. Kjo rrit ngarkim në infrastrukturë krahasuar me shërbimin e fajllëve statikë.
2. **TTFB i lartë nën ngarkim:** Nëse serveri është i ngarkuar ose thirrjet API janë të ngadalta, TTFB rritet dhe kjo ndikon negativisht te LCP.
3. **Kompleksiteti i Hydration:** Procesi i hydration-it ende kërkon dërgimin e JavaScript-it tek klienti. Nëse bundle-i JavaScript është i madh, koha deri në interaktivitet (Time to Interactive) mbetet e lartë.
4. **Gjendja e ndryshme midis serverit dhe klientit:** Nëse kodi React prodhon rezultate të ndryshme gjatë renderimit në server dhe gjatë hydration-it (p.sh., përdorimi i window ose Date.now()), krijohen gabime hydration të vështira për t'u diagnostikuar.

Për aplikacione ku të dhënat ndryshojnë shpesh dhe janë specifike për çdo përdorues (p.sh., panele administrimi, aplikacione bankare), SSR është zgjidhja e përshtatshme. Për përmbajtje që ndryshon rrallë, Static Site Generation ofron performancë superiore, siç do të trajtohet në kapitullin vijues.

5. Static Site Generation (SSG) – Gjenerimi Statik i Faqeve

5.1. Çfarë është SSG

Siç paraqitet në figurën e mësipërme, në arkitekturën Static Site Generation (SSG) procesi i gjenerimit të faqes ndahet në dy faza kryesore. Në fazën e parë, gjatë procesit të ndërtimit (build time), serveri merr të dhënat e nevojshme, gjeneron HTML-in dhe fajllat përkatës statikë, dhe i publikon ato në një **CDN** - një grup serverash me të vendosur në lokacione të ndryshme rreth botës. Në fazën e dytë, kur përdoruesi kërkon faqen, CDN-ja shërben menjëherë përmbajtjen e parakompiluar pa pasur nevojë për renderim në kohë reale në server. Kjo qasje redukton ndjeshëm vonesat gjatë ngarkimit fillestar dhe mundëson shpërndarje shumë efikase të përmbajtjes në shkallë globale.

Static Site Generation (SSG) është modeli ku HTML-i gjenerohet një herë gjatë procesit të ndërtimit (build time) dhe ruhet si fajlla statikë, të gatshëm për t'u shërbyer drejtpërdrejt nga një CDN (Content Delivery Network) ose server statik. Kur një përdorues kërkon faqen, ai merr menjëherë HTML-in e parakompiluar, pa ekzekutim logjike në server në kohë reale (GeeksforGeeks, 2026).

Burimi: Fotografi e autorit

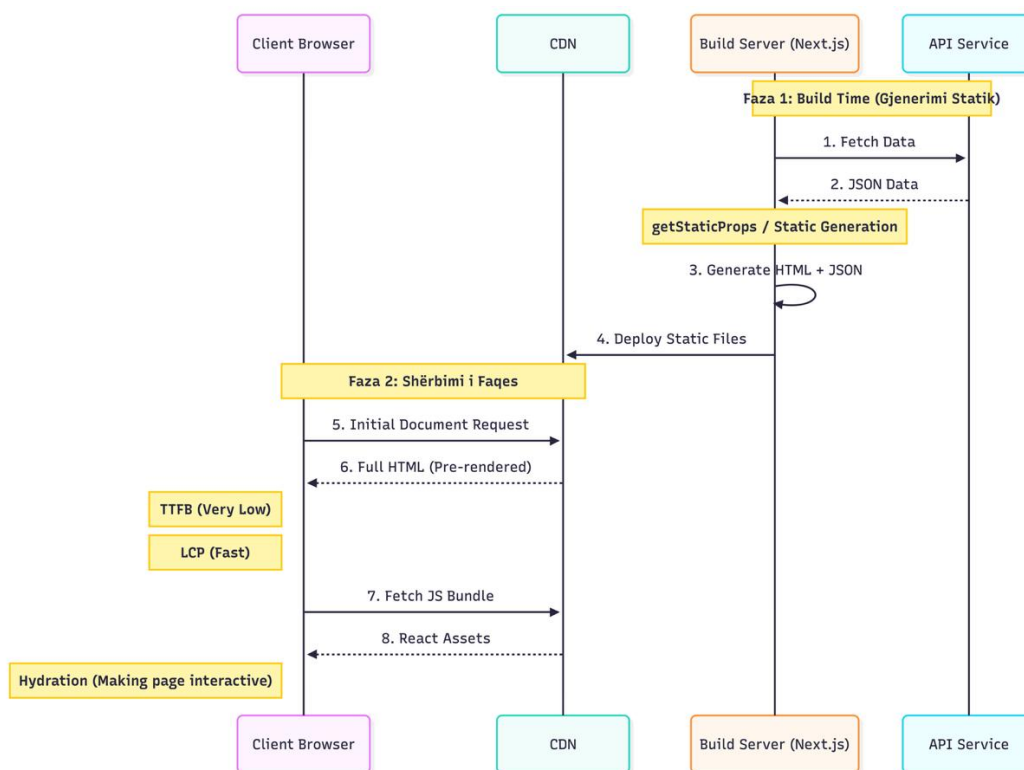


Figure 5 SSG - Procesi i SSG në Fazën e Ndërtimit dhe të Shërbimit

5.2. Si funksionon SSG me Next.js

Në Next.js, SSG implementohet nëpërmjet funksionit **getStaticProps (Pages Router)** ose nëpërmjet Server Components me cache-im statik (App Router). Gjatë komandës "next build", Next.js ekzekuton getStaticProps për çdo faqe, merr të dhënat e nevojshme dhe gjeneron fajllat HTML përkatës. Rezultati final është një grup fajllash HTML+JSON të gatshëm për deployment. Për faqe me rrugë dinamike (p.sh., /blog/[slug]), funksioni generateStaticParams specifikon listën e rrugëve që duhen gjeneruar paraprakisht. Faqe që nuk janë gjeneruar paraprakisht mund të gjenerohen "on-demand" herën e parë që kërkohen dhe pastaj të ruhen (Incremental Static Regeneration).

5.3. ISR: Incremental Static Regeneration

ISR është inovacion i Next.js që kombinon avantazhet e SSG me aftësinë e rigjenerimit periodik. Duke specifikuar një parametër "revalidate" (në sekonda), faqja statike rigjenerohet në background kur mbetet pas kohe, duke shërbyer versionin e ri të ardhshëm klientëve pasardhës.

Kjo lejon aplikacione me të dhëna gjysmë-dinamike (blog, katalog produkte, faqe lajmesh) të gëzojnë shpejtësinë e plotë të shërbimit statik pa sakrifikuar freskinë e përmbajtjes (Vercel, 2026).

5.4. SSG dhe Web Vitals

SSG ofron performancën e Web Vitals afër optimale:

LCP: Duke qenë se HTML-i shërbehet nga CDN pa asnjë llogaritje në server, TTFB është minimal (shpesh nën 50ms) dhe elementi LCP zbulohet e renderohet shumë shpejt.

CLS: Meqë HTML-i është i plotë dhe statik, nuk ka ngarkime asinkrone që shkaktojnë zhvendosje paraqitjeje.

INP: Pas hydration-it, interaktiviteti varet nga sasia e JavaScript-it dërguar te klienti. SSG vetëm nuk adreson INP; nevojiten teknikat e optimizimit të bundle-it dhe code-splitting të cilat trajtohen në Kapitullin 7 (Osmani, a.d.).

5.5. Krahësimi SSR vs SSG vs CSR

Modeli	TTFB	LCP	I skalueshëm	Të dhëna të freskëta
CSR	<i>I ulët</i>	<i>I lartë</i>	<i>E lartë</i>	<i>Gjithmonë</i>
SSR	<i>Mesatar</i>	<i>I ulët</i>	<i>Mesatare</i>	<i>Gjithmonë</i>
SSG	<i>Shumë ulët</i>	<i>Shumë ulët</i>	<i>Shumë e lartë</i>	<i>Sipas revalidate</i>
SSG + ISR	<i>Shumë ulët</i>	<i>Shumë ulët</i>	<i>Shumë e lartë</i>	<i>Periodike</i>

6. Streaming dhe React Suspense

6.1. Kufizimi i SSR klasik: Bllokimi "All-or-Nothing"

SSR tradicional ka një kufizim strukturor: serveri duhet të ekzekutojë të gjithë logjikën dhe të marrë të gjitha të dhënat përpara se të dërgojë çfarëdo HTML te klienti. Nëse një komponent kërkon të dhëna nga një API e ngadaltë, i gjithë renderimi bllokohet derisa ajo thirrje të përfundojë.

Kjo mund të rrisë TTFB dhe të anulojë avantazhet e SSR.

6.2. Streaming SSR me React 18

React 18 prezantoi Streaming SSR, i cili lejon serverin të dërgojë HTML-in në copa (chunks) ndërkohë që komponentët e ndryshëm renderohen. Shfletuesi fillon të shfaqë dhe të parsojë HTML-in e marrë ndërkohë që serveri vazhdon të gjenerojë pjesët e mbetura. Kjo arrihet duke kombinuar:

- **React.Suspense:** Komponenti Suspense i React lejon specifikimin e një "fallback" (p.sh., skeleton screen) ndërkohë që fëmijët e tij po ngarkohen. Gjatë Streaming SSR, serveri dërgon fallback-un menjëherë dhe më pas zëvendëson me përmbajtjen reale sapo ajo të jetë gati.

- **renderToPipeableStream / renderToReadableStream**: API-të e reja të React që mbështesin streaming-un nëpërmjet Node.js Streams ose Web Streams (Meta, n.d.; Meta, 2022).

6.3. Selective Hydration

Krahas Streaming SSR, React 18 mbështet Selective Hydration: komponentët hidratohen në mënyrë selektive dhe me prioritet bazuar në ndërveprimet e përdoruesit. Nëse përdoruesi klikon në një komponent që ende nuk është hidratuar, React i jep prioritet atij komponenti. Kjo redukton ndjeshëm efektin negativ të hydration-it mbi INP, pasi fija kryesore (main thread) nuk bllokohet nga hidratimi i njëhershëm i të gjithë faqes.

6.4. React Server Components (RSC)

Me App Router-in e Next.js 13+, React Server Components (RSC) ndryshojnë thellësisht modelin e renderimit. Server Components ekzekutohen ekskluzivisht në server dhe nuk dërgojnë asnjë kod JavaScript te klienti. Vetëm Client Components (të shënuara me "use client") dërgojnë JavaScript te shfletuesi.

Ky model redukton ndjeshëm madhësinë e bundle-it JavaScript të dërguar te klienti, duke ndikuar pozitivisht tek INP, pasi main thread ka më pak kod për të ekzekutuar. Logjika e rëndë si parsing i Markdown-it, aksesit i bazës të dhënash, ose thirrjet ORM mbeten tërësisht në server.

6.5. Ndikimi i Streaming dhe RSC në Web Vitals

LCP: Me Streaming SSR, HTML-i fillon të mbërrijë menjëherë (TTFB i ulët), ndërkohë komponentët e ngadaltë renderohen progresivisht. Elementi LCP mund të dërgohet i pari, para komponentëve dytësorë.

INP: Selective Hydration dhe RSC ulin sasinë totale të JavaScript-it që ekzekutohet gjatë ngarkimit, duke liruar main thread-in që të përgjigjet ndaj ndërveprimeve të përdoruesit me vonesë minimale.

CLS: Fallback-u i Suspense (skeleton screens me dimensione të rezervuara) parandalon zhvendosjet e papritura të paraqitjes kur të dhënat mbërrijnë.

7. Modularizimi dhe Optimizimi i JS Bundle

7.1. Problemi i Bundle-it të Madh

Në aplikacionet React të shkallës së lartë, bundle-i JavaScript rritet me çdo varësi dhe komponent të ri. Një bundle i vetëm i madh (monolithic bundle) ka pasoja serioze:

- Koha e shkarkimit rritet, veçanërisht në rrjete mobile.
- Koha e parsing dhe kompajlimit të JavaScript rritet, bllokojnë main thread.
- Kodi i faqeve që nuk janë vizituar ende ngarkohet panevojshëm.

7.2. Code Splitting

Code splitting është teknika e ndarjes së bundle-it në copa më të vogla që ngarkohen vetëm kur nevojiten. React dhe bundler-ët modernë (Webpack, Vite, Turbopack) e mbështesin natyrshëm nëpërmjet importit dinamik:

```
const Dashboard = React.lazy(() => import('./Dashboard'));
```

Next.js aplikon code splitting automatik bazuar në rrugë: çdo faqe gjeneron chunk-un e saj JavaScript, dhe vizitorët nuk shkarkojnë kurrë kodin e faqeve të pa-vizituara (Vercel, 2026).

7.3. Lazy Loading i Komponentëve

React.lazy() dhe Suspense lejojnë ngarkimin e vonuar të komponentëve të rëndë (p.sh., editorë teksti, grafike, harta). Komponenti ngarkohet vetëm kur hyn në viewport ose kur aktivizohet nga ndërveprim i përdoruesit. Kjo redukton drastikisht JavaScript-in e ekzekutuar gjatë ngarkimit fillestar (Meta, n.d.).

7.4. Tree Shaking

Tree shaking është procesi i eliminimit automatik të kodit të pa-përdorur gjatë procesit të ndërtimit. Bundler-ët modernë analizojnë grafikun e importeve dhe eksporteve dhe heqin modulet dhe funksionet që nuk thirren kurrë. Kjo kërkon:

- Përdorimin e sintaksës ES Modules (import/export) dhe jo CommonJS (require).
- Shmangien e importeve "barrel" të panevojshme (p.sh., import { everything } from 'huge-library').
- Preferimin e librarive me mbështetje të mirë të tree shaking (p.sh., date-fns mbi moment.js).

7.5. Optimizimi i Varësive të Palës së Tretë

Libraritë e palës së tretë shpesh përbëjnë pjesën më të madhe të bundle-it. Strategjitë e optimizimit përfshijnë:

- Importim i pjesshëm: Importimi vetëm i moduleve që nevojiten.
- Alternativa më të lehta: Për shembull, day.js (2KB) si alternativë ndaj moment.js (67KB).
- Ngarkimi i vonuar i skripteve të palës së tretë: Analytics, chat widgets dhe skripte reklamash duhet ngarkuar pas ngarkimit kryesor me atributun "defer" ose nëpërmjet Next.js Script me strategy="lazyOnload".
- Analizimi i bundle-it: Tools si @next/bundle-analyzer apo webpack-bundle-analyzer vizualizojnë përbërjen e bundle-it dhe identifikojnë burimet e fryerjes.

7.6. Strategjia e Optimizimit të Imazheve

Imazhet janë shpesh faktori kryesor për LCP të vonuar. Next.js ofron komponentin `<Image>` i cili automatikisht:

- Konverton imazhet në formate moderne (WebP, AVIF).
- Aplikon lazy loading për imazhet jashtë viewport-it.
- Rezervon hapësirën (width/height) për të parandaluar CLS.
- Ofron imazhe të ndryshme dimensionesh sipas madhësisë së ekranit (responsive srcset). (Vercel, n.d.).

8. Studimi i rastit – Migrimi nga CSR ne SSR/SSG

8.1. Përshkrimi i Skenarit

Ky studim rasti analizon një aplikacion React ekzistues, të ndërtuar me React 19 dhe Vite dhe arkitekturë CSR, që shërben si platformë katalog produktesh me 620 produkte, të trajtuara nga 3 ruta (/ , /product/:id, /about).

Karakteristikat e aplikacionit fillestar:

- Framework: React 19 + Vite
- Renderimi: CSR i plotë
- Bundle kryesor JS: ~374 KB (gzip: 123 KB)
- Numri i faqeve: ~620 produkte te trajtuara në tre ruta

8.2. Matjet Fillestare (Para Migrimit)

Matjet u kryen duke përdorur Lighthouse me throttling real (jo të simuluar) përmes Chrome DevTools Protocol — rrjet 4G i ngadaltë (150ms RTT, ~1.6 Mbps) dhe CPU 4x më i ngadaltë, mesatarja mbi 3 xhirime për faqe. Rezultatet mesatare:

Metrika	Vlera fillestare	Vlerësimi
LCP	5.18 sekonda	Dobët (> 4.0s)
INP	I pamatshëm në lab*	Shih shënimin 8.4
CLS	0.238	Nevojitet Përmirësim (0.1-0.25)
TTFB	~1 ms (lokal)*	Jo krahasues — shih 8.5
Bundle Size (main)	374 KB (gzip: 123 KB)	—
First Contentful Paint	2.11 sek.	Nevojitet Përmirësim

Problemet e identifikuara:

- LCP i lartë: Hero image dhe lista e produkteve renderohen pas ekzekutimit të JS dhe kërkesës API (zinxhir varësish 4-hëshesh).
- CLS 0.238: Produktet shfaqen me vonesë dhe zhvendosin elemente tipografike.
- Bundle i madh: Të gjitha faqet ndajnë të njëjtin bundle monolit.

8.3. Strategjia e Migrimit

Aplikacioni u migrua në Next.js 15 me App Router duke ndjekur hapat:

1. Migrimi i Strukturës: Faqet u transferuan në direktorinë /app; komponentët u etikëtuan si Server Components (parazgjedhja) ose Client Components ("use client") sipas nevojës për interaktivitet.
2. Faqja Kryesore me SSG+ISR: Faqja kryesore me listën e produkteve të rekomanduara u konvertua në SSG me revalidate = 3600 sekonda (1 orë). Lista merret gjatë build time dhe rigjenerohet çdo orë.
3. Faqet e Produkteve me SSG: 500 faqet e produkteve u gjeneruan statikisht me generateStaticParams. Faqet e produkteve të reja shtuar pas deploy-it nderohen me ISR me revalidate = 86400 (24 orë).
4. Optimizimi i Imazheve: Hero image dhe imazhet e produkteve u zëvendësuan me komponentin next/image me prioritet={true} për hero image (parandalon lazy loading për LCP element).
5. Ndarje automatike route-based + <Suspense> real për streaming të "Rekomandimeve"

6. Skeleton Screens: U shtuan skeleton screens me dimensione fikse për të parandaluar CLS gjatë ngarkimit asinkron.

8.4. Matjet Pas Migrimit

METRIKA	PARA MIGRIMIT	PAS MIGRIMIT	NDRYSHIMI
LCP	5.18 sek.	1.39 sek.	-73%
TBT (PROXY)*	~0 ms	~15 ms	—
CLS	0.238	0.000	-100%
TTFB	~1 ms*	~2 ms*	—
BUNDLE (MAIN)	374 KB	~102 KB	-73%
FCP	2.11 sek.	1.33 sek.	-37%

Metrikat e matura drejtpërdrejt — LCP, CLS, dhe metrika plotësuese FCP — kaluan nga kategoria "Dobët"/"Nevojitet Përmirësim" në kategorinë "Mirë" sipas pragut të Google. INP nuk mund të matet drejtpërdrejt në kushte laboratorik (kërkon të dhëna reale ndërveprimi/RUM); Total Blocking Time, përdorur si proxy laboratorik, mbeti nën 20ms në të dyja variantet. TTFB nuk qe krahasues në këtë mjedis lokal, pasi të dy serverët përgjigjen në më pak se 10ms pavarësisht arkitekturës (shih 8.5).

8.5. Dëshmi Vizuale nga DevTools (Chrome Performance)

Për të plotësuar matjet e bazuara në Lighthouse (seksionet 8.2 dhe 8.4), më poshtë paraqiten dy pamje nga paneli "Performance → Live metrics" i Chrome DevTools, i cili mat metrikat Core Web Vitals drejtpërdrejt në shfletues gjatë ngarkimit real të faqes (throttling: Slow 4G).

Burimi: Fotografia e autorit

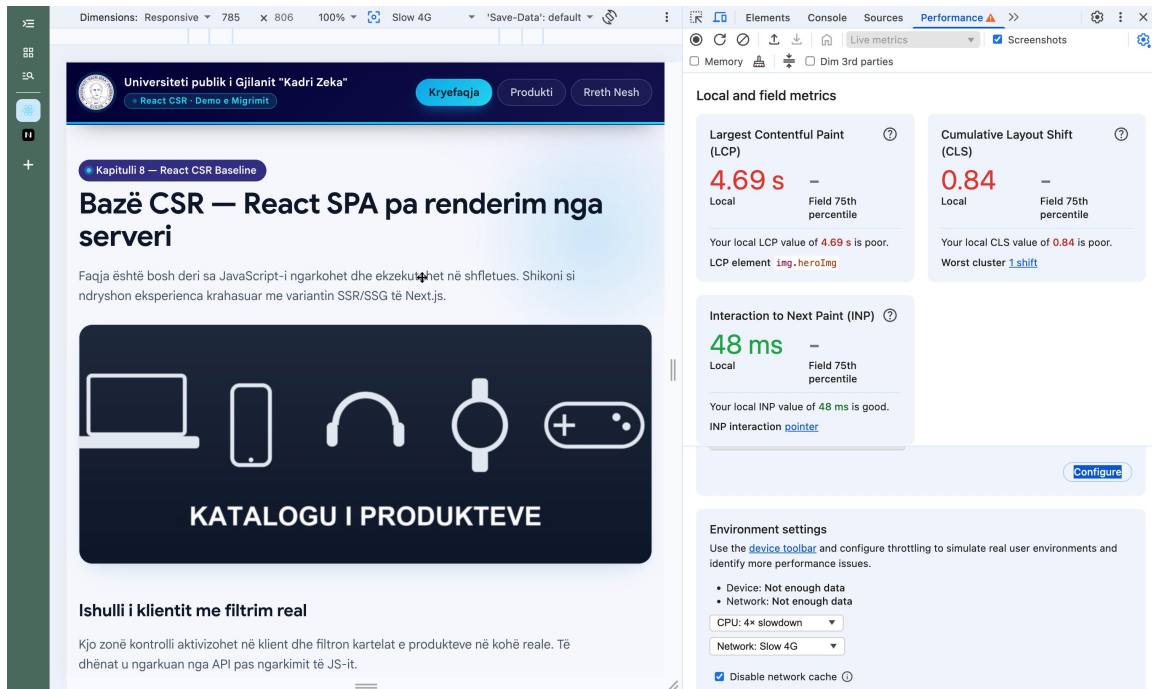


Figure 6 CSR – Metrikat live nga Chrome DevTools (para migrimit)

Burimi: Fotografia e autorit

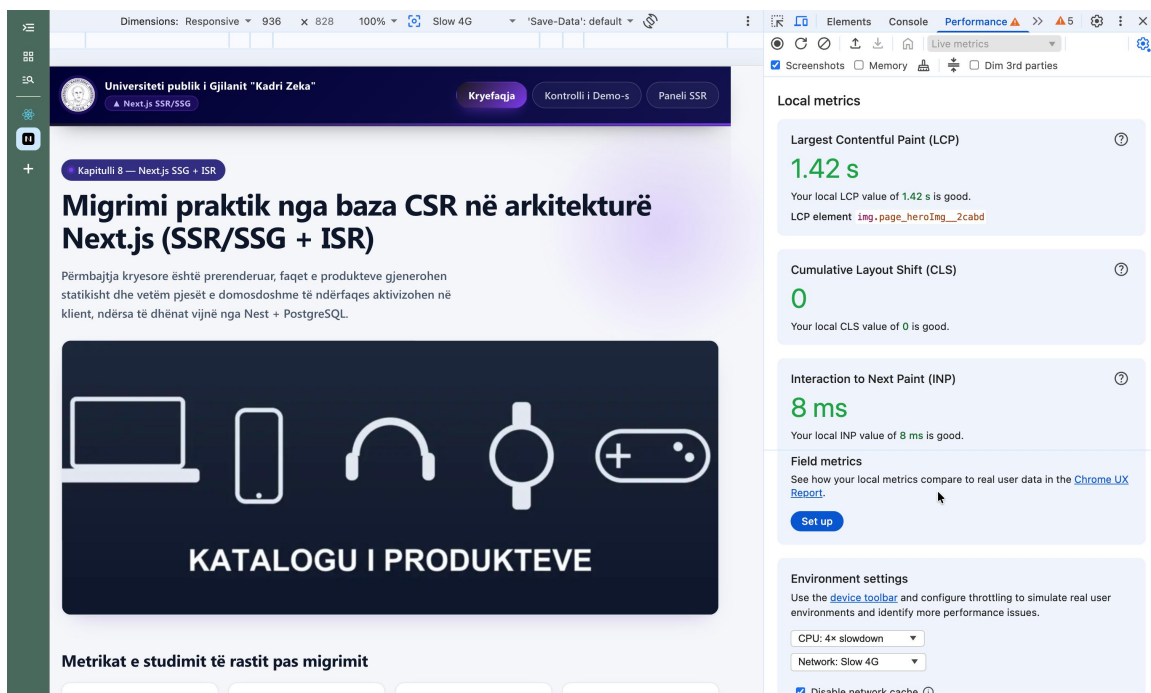


Figure 7 Next.js – Metrikat live nga Chrome DevTools (pas migrimit)

Në rastin CSR, LCP rezulton 4.69 sekonda ("Dobët") dhe CLS 0.84 ("Dobët") — CLS këtu është më i lartë se në Tabelën 8.2 (0.238), pasi kjo pamje përfshin edhe zhvendosjen shtesë të shkaktuar nga vetë ndërveprimi (klikimi) i kryer për të regjistruar INP-në. INP rezulton 48 ms ("Mirë"), i matur pas një ndërveprimi real (pointer). Në rastin Next.js, nën të njëjtat kushte throttling (CPU: 4x slowdown, Network: Slow 4G — identike me kapjen e CSR-së), LCP rezulton 1.42 sekonda ("Mirë"), CLS 0 ("Mirë"), dhe INP 8 ms ("Mirë"). Meqë të dyja pamjet janë kapura tani me kushte throttling identike, krahasimi është drejtpërdrejt dhe simetrik.

8.6. Vërejtje metodologjike

Matjet u kryen në mjedis laborator (Chrome DevTools throttling: 150ms RTT, ~1.6 Mbps, CPU 4x, mesatare e 3 xhirimeve për faqe) dhe jo në trafik real. Matjet në laborator janë reproduktibile dhe krahasuese, por nuk zëvendësojnë plotësisht matjet e fushës (Chrome User Experience Report — CrUX) pasi faktorë si shpërndarja gjeografike e përdoruesve, tipet e pajisjeve dhe cilësia e rrjetit ndryshojnë. Dy kufizime shtesë vlejné për këtë studim rasti: (1) INP nuk u mat drejtpërdrejt, pasi Lighthouse në modalitet laborator nuk gjeneron një vlerë INP reale — kjo kërkon ndërveprim faktik të përdoruesit, i disponueshëm vetëm përmes RUM; (2) TTFB doli pothuajse identik (nën 10ms) për të dy arkitekturat, sepse audit-i "server-response-time" i

Lighthouse izolon kohën e përpunimit në server nga vonesa e rrjetit, dhe backend-i lokal (NestJS + PostgreSQL) përgjigjet shpejt pavarësisht throttling-ut të rrjetit; diferenca reale e TTFB-së mes arkitekturave do të shfaqej vetëm në një vendosje (deployment) reale me distancë rrjeti faktike (p.sh. CDN statik kundrejt një serveri origjinal). Implementimi real do të kërkonte monitorim të vazhdueshëm nëpërmjet Google Search Console dhe RUM (Real User Monitoring) tools. (Google, 2024)

9. Përfundimi dhe rekomandime

9.1. Përfundimi

Ky punim demonstroi se performanca e aplikacioneve React të shkallës së lartë nuk është thjesht çështje e shkrimit të kodit efikas, por para së gjithash çështje e zgjedhjes dhe aplikimit të saktë të arkitekturës.

Analiza e aplikacioneve SPA/CSR tregoi se kufizimet strukturore të këtij modeli — varësia e plotë nga JavaScript-i i klientit, zinxhiri i gjatë i varësive për LCP, bllokimi i main thread gjatë hydration-it dhe ndjeshmëria ndaj gabimeve të ekzekutimit — shkaktojnë degradim të matshëm të Web Vitals dhe të përvojës së përdoruesit, veçanërisht në pajisje mobile dhe rrjete të ngadalta.

Metrikat Core Web Vitals — LCP, INP dhe CLS — ofrojnë një kornizë të unifikuar dhe sasiore për vlerësimin e cilësisë së ndërfaqes. Lidhja mes arkitekturës dhe këtyre metrikave është e drejtpërdrejtë dhe e matshme: zgjedhja e modelit të renderimit (CSR, SSR, SSG) dhe strategjitë e optimizimit të bundle-it ndikojnë drejtpërdrejt në secilën prej këtyre metrikave.

Modelet SSR dhe SSG, të implementuara nëpërmjet Next.js, adresojnë problemet strukturore të CSR duke zhvendosur peshën e renderimit drejt serverit ose procesit të ndërtimit. Streaming SSR me React 18 dhe React Server Components reduktojnë JavaScript-in e dërguar te klienti dhe optimizojnë mënyrën se si hydration-i ndodh, duke përmirësuar INP. Teknikat e code splitting, lazy loading dhe optimizimit të imazheve plotësojnë zgjedhjen arkitekturore me optimizime në nivel komponenti.

Studimi i rastit tregoi se migrimi i planifikuar nga CSR në Next.js me SSG solli përmirësime të matshme dhe të riprodhueshme: -73% LCP, -37% FCP, -100% CLS (nga 0.238 në 0.000), dhe -73% madhësi bundle-i — duke i kaluar metrikat e matura direkt nga kategori negative në "Mirë". INP mbetet një përmirësim i pritshëm teorikisht (bazuar në reduktimin e JavaScript-it të ekzekutuar dhe në Selective Hydration/RSC, shih Kapitulli 6), pasi nuk mund të matet drejtpërdrejt në kushte laborator.

9.2. Rekomandime Praktike

Bazuar në analizën e realizuar, rekomandimet kryesore për zhvilluesit dhe arkitektët e aplikacioneve React të shkallës së lartë janë:

1. Vlerëso nevojat e të dhënave para zgjedhjes së modelit të renderimit: Për përmbajtje statike ose gjysmë-statike, SSG me ISR është zgjedhja optimale. Për të dhëna specifike per-user ose shumë dinamike, SSR ose CSR me optimizime janë të përshtatshme.
2. Matje para optimizimit: Mos optimizo pa matje. Lighthouse, Chrome DevTools Performance panel dhe WebPageTest ofrojnë të dhëna konkrete mbi bottleneck-ët reale.
3. Prioritizo buxhetin JavaScript: Synoji < 200KB JavaScript total (gzip) për ngarkimin fillestar. Analizoji bundle-in rregullisht me @next/bundle-analyzer.
4. Optimizo LCP element-in eksplicitisht: Identifiko elementin LCP të çdo faqeje (zakonisht hero image ose titull kryesor) dhe siguro se ai ngarkohet me prioritet (preload, priority prop në next/image).
5. Mbro main thread-in: Shpërndaj logjikën e rëndë në Server Components ose Web Workers. Shmang re-renderime të panevojshme me React.memo dhe useMemo në vendet e duhura.
6. Testo performancën në kushte reale: Mjedisi i zhvillimit nuk pasqyron pajisjen mobile me rrjet 4G. Inkorporo Lighthouse CI ose matje RUM (p.sh., Vercel Analytics, Google CrUX) si pjesë të pipeline-it të deployment-it.

9.3. Perspektiva dhe punë e ardhshme

Fusha e arkitekturimit të aplikacioneve React po evoluon me shpejtësi. Dy drejtime kryesore meritojnë vëmendje në punë të ardhshme: Partial Prerendering (PPR), funksion eksperimental në Next.js 14+, kombinon HTML-in statik dhe dinamik në të njëjtën faqe: korniza e faqes renderohet statikisht ndërkohë "vrima" (slots) dinamike streamini SSR nga serveri (Vercel, n.d.). Kjo prometon të kombinojë TTFB të ulët të SSG me freskinë e SSR.React Compiler (i prezantuar në React 19), e cila optimizon automatikisht re-renderimet pa pasur nevojë për memo-izim manual, premtan të thjeshtojë ndjeshëm optimizimin e komponentëve dhe të rrisë INP pa ndërhyrje manuale nga zhvilluesi (Meta, 2024).

Referencat

- [1] GeeksforGeeks. (2026, January 22). *What is Single Page Application?* From geeksforgeeks: <https://www.geeksforgeeks.org/javascript/what-is-single-page-application/>
- [2] Vercel. (2026, May 28). *Server Components*. From Next.js: <https://nextjs.org/docs/app/building-your-application/rendering/server-components>
- [3] Vercel. (2026, May 28). *Static Site Generation*. From Next.js: <https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation>
- [4] Vercel. (2026, May 28). *Incremental Static Regeneration*. From Next.js: <https://nextjs.org/docs/pages/building-your-application/data-fetching/incremental-static-regeneration>
- [5] Google. (2020, May 4). *Web Vitals*. From web.dev: <https://web.dev/articles/vitals>
- [6] Osmani, A. (a.d.). *Rendering Patterns*. From patterns: patterns.dev/react/rendering-patterns
- [7] Google. (2025). *Largest Contentful Paint (LCP)*. From web.dev: <https://web.dev/articles/lcp>
- [8] Google. (2025). *Interaction to Next Paint (INP)*. From web.dev: <https://web.dev/articles/inp>
- [9] Google. (2025). *Time to First Byte (TTFB)*. From web.dev: <https://web.dev/articles/ttfb>
- [10] Meta. (n.d.). *Render and Commit*. From React Documentation: <https://react.dev/learn/render-and-commit>
- [11] Meta. (n.d.). *Suspense*. From React Documentation: <https://react.dev/reference/react/Suspense>
- [12] Meta. (n.d.). *React.lazy*. From React Documentation: <https://react.dev/reference/react/lazy>
- [13] Meta. (2022, March 29). *React 18: New features*. From React Blog: <https://react.dev/blog/2022/03/29/react-v18>
- [14] Vercel. (n.d.). *Data Fetching: getServerSideProps*. From Next.js Documentation: <https://nextjs.org/docs/pages/building-your-application/data-fetching/get-server-side-props>
- [15] Vercel. (n.d.). *Image Optimization*. From Next.js Documentation: <https://nextjs.org/docs/app/building-your-application/optimizing/images>
- [16] Vercel. (n.d.). *Partial Prerendering*. From Next.js Documentation: <https://nextjs.org/docs/app/building-your-application/rendering/partial-prerendering>
- [17] Google. (2024, February 08). *Chrome User Experience Report*. From Chrome Developers: <https://developer.chrome.com/docs/crux>
- [18] Google. (2023). *Cumulative Layout Shift (CLS)*. From web.dev: <https://web.dev/articles/cls>
- [19] Meta. (2024, October 21). *React Compiler*. From React Documentation: <https://react.dev/learn/react-compiler>