Optimization of Content Delivery in Web Applications

MASTER THESIS

Tobias Schmid

Submitted on 27 March 2026

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Andreas Kaufmann PhD
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 27 March 2026

License

Erlangen, 27 March 2026

Acknowledgements

I express my sincere gratitude to Andreas for his invaluable guidance, unwavering support, and expert advice throughout this journey.

In addition, I acknowledge the use of ChatGPT in improving the structure and language quality of this thesis.

Finally, I would like to thank all the proofreaders for their valuable time and their contribution to improving this thesis.

Abstract

This thesis improves content delivery for QDAcity, a localized React single-page application, by enabling incremental localization alongside ongoing feature development and systematic management of the translation status. It introduces a selective build-time prerendering pipeline for a subset of routes, while keeping dynamic pages client-rendered, improving build performance without sacrificing user experience. Localization is refactored into route-scoped namespaces with on-demand message loading, caching, deduplication, and predictive prefetching, so new languages can be shipped incrementally with minimal payload and fast navigation. The translation lifecycle is integrated via Weblate and CI automation, treating each namespace as stateful (missing, translated, approved) to govern publication, fallbacks, and user-facing disclosure for unreviewed translations. The system is evaluated against the baseline in terms of build efficiency, Core Web Vitals, translation bundle size, and correctness, providing a scalable workflow for multilingual web applications.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence

API Application Programming Interface

CAQDAS Computer-Assisted Qualitative Data Analysis Software

CDN Content Delivery Network

CI Continuous Integration

CLS Cumulative Layout Shift

CSR Client Side Rendering

CWV Core Web Vitals

CrUX Chrome User Experience Report

DOM Document Object Model

FID First Input Delay

HTML HyperText Markup Language

INP Interaction to Next Paint

JSON JavaScript Object Notation

LCP Largest Contentful Paint

MR Merge Request

SEO Search Engine Optimization

SPA Single Page Application

SSG Static Site Generation

SSR Server Side Rendering

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

1 Introduction

Content delivery is the engineering discipline of delivering web content quickly and reliably and selecting an appropriate delivery and rendering strategy for the user context (e.g., language, device constraints, or network conditions). Consider a user opening a website on a phone while commuting: the objective may be to check a price, read a paragraph, or find a feature, and the main content is expected to load quickly and seamlessly. When it does not, the degradation becomes apparent: the page may appear unstable, users may lose context, and sessions may end before the intended task is completed. A common reaction can be to refresh or retry, but this can exacerbate the situation by restarting network requests, re-downloading resources, and resetting in-progress state. On mobile networks with limited throughput, reloading can further increase perceived waiting time and network data usage. This everyday situation illustrates why *content delivery* matters.

Modern web systems intensify this challenge through two simultaneous trends. First, web experiences have become richer and more application-like: many sites ship large JavaScript bundles, deliver high-resolution media, and rely on interactive and stateful client-side behavior. In practice, many web pages are closer to client applications than to documents. Second, expectations have tightened: responsiveness and visual stability are rewarded not only by users, but also by discovery and distribution platforms, most notably by search engines. *Google* recommends achieving good Core Web Vitals (CWV) (metrics that measure real-world loading performance, interactivity, and visual stability) and notes that these page experience signals align with what its core ranking systems seek to reward.[1] These signals are not abstract: the Chrome User Experience Report (CrUX), which aggregates field data from real *Chrome* users, makes the performance gaps visible at scale.

CrUX data indicates that a substantial share of origins do not achieve good values across all three CWV. In the CrUX November 2025 dataset[2], only 54.6%

[1] https://developers.google.com/search/docs/appearance/core-web-vitals accessed 08.02.2026

[2] https://developer.chrome.com/docs/crux/release-notes accessed 07.02.2026

of origins achieved good values across all three CWV. Additionally, interactivity differs between desktop and mobile, with 97% of sites achieving a good Interaction to Next Paint (INP) rating on desktop devices, compared to 74% on mobile devices (Zigisova & Akrap, 2024). This indicates that performance optimization is not merely a "nice-to-have" but a recurring engineering problem that impacts user experience and discoverability.

Furthermore, the deployment context is shifting: systems that historically ran on-premises are modernized and moved into cloud or hybrid cloud setups, which changes how content is produced, cached, and delivered globally. *Flexera* reports that migration of workloads to the cloud is among the top initiatives for 58% of organizations (Flexera, 2024), and *Gartner* predicts that 90% of organizations will adopt a hybrid cloud approach by 2027 (Gartner, 2024). Accordingly, this thesis focuses on improving how localized content is generated and packaged so that it can be cached and delivered globally in cloud and hybrid-cloud contexts.

In the baseline architecture of *QDAcity*, the generation of statically prerendered HyperText Markup Language (HTML) relies on the crawler-based library *react-snap*, which is no longer actively maintained[3] and is build-time intensive as route and language coverage increases. In addition, translations are managed as a monolithic translation catalog, which couples runtime payload to overall application size and complicates incremental rollout. These characteristics limit scalability and make controlled, incremental language publication difficult. Consequently, an approach is required that aligns prerendering and localization more directly with routes and publication status.

Against this background, this thesis designs and implements a content-delivery solution for a modern, localized web application under two practical constraints: the system must remain maintainable for ongoing feature development, and localization must support a continuous translation workflow without requiring complete language coverage upfront. The solution reduces critical-path runtime work and payload while keeping language-specific Uniform Resource Locators (URLs) stable to support predictable navigation and Search Engine Optimization (SEO).

This thesis's approach is implemented and assessed within the QDAcity web application. QDAcity is a browser-based Computer-Assisted Qualitative Data Analysis Software (CAQDAS) application that supports qualitative research workflows such as organizing textual data, applying and managing codes, and collaborating on analysis projects. As a content- and interaction-heavy single-page application, QDAcity combines frequent navigation across many routes with a strong need for fast initial rendering, predictable user experience, and search-engine discoverability of public-facing documentation pages. At the same time,

[3]https://github.com/stereobooster/react-snap (last non-documentation update in November 2022: 14 Nov 2022)

QDAcity is developed and operated with a multilingual target audience in mind, making continuous localization and incremental translation rollout a practical requirement rather than an optional enhancement.

The thesis presents four complementary components:

- **Generate static content for suitable routes:** Selected routes are prerendered using SSG to improve initial content availability and reduce client-side work on the critical path, while preserving a hybrid model for pages that require client-side rendering.
- **Namespace translations by route:** Translation resources are partitioned into route-associated namespaces. At runtime, only the namespaces required for the active route are loaded, reducing unnecessary transfer and decoupling translation payload growth from overall application size.
- **Integrate Weblate into the development lifecycle:** A translation workflow is integrated into development through automation and tooling (e.g., configuration, synchronization scripts, and Continuous Integration (CI) checks). This provides traceability and quality control for localization changes without coupling the runtime loading mechanism to the translation platform.
- **Enable incremental translation with explicit states and governed fallback:** Languages are introduced and expanded step-by-step using explicit translation states (e.g., *translated* vs. *approved*). Deterministic runtime behavior for partially translated routes is enforced via a controlled fallback policy and user-facing disclaimers that communicate incomplete or unapproved localization while keeping language-specific URLs stable.

Chapter 2 reviews user-centric web performance, rendering strategies, search engine optimization, and continuous localization with *Weblate*. Chapter 3 derives functional and non-functional requirements from these foundations. Chapters 4 and 5 present the architecture and implementation of the SSG- and namespace-based approach, including Weblate integration and the runtime loading strategy for route-aligned translation resources. Chapter 6 evaluates the implemented approach against the baseline with respect to performance, scalability, and correctness. Chapter 7 concludes with a summary of contributions, limitations, and future work.

2 Foundations and Background

This chapter summarizes the concepts required to discuss content delivery and localization in modern web applications. It introduces CWV as user-centric performance signals, reviews common rendering strategies and their trade-offs, and outlines continuous localization with Weblate. The chapter establishes the terminology and baseline assumptions used to derive requirements, motivate architectural decisions, and interpret evaluation results in later chapters.

2.1 Core Web Vitals

The CWV are a set of user-centric performance metrics defined by Google to quantify key dimensions of perceived page experience for web pages: loading performance, responsiveness, and visual stability. The current CWV metrics consist of Largest Contentful Paint (LCP), INP, and Cumulative Layout Shift (CLS).[1] Google notes that CWV are used by its ranking system as part of broader page experience considerations, while relevance remains the primary ranking factor.[2]

CWV are designed to be judged based on field data, i.e. real user measurements collected under real devices and network conditions. Google recommends evaluating CWV at the 75^{th} percentile of page loads and considering differences across device classes such as mobile and desktop.[3] In many tools, CWV field metrics are reported based on the CrUX report, which reports aggregated user experience over a monthly period.[4]

Table 2.1 summarizes the commonly used CWV thresholds based on values provided by Google.

[1] https://developers.google.com/search/docs/performance/core-web-vitals accessed: 28.12.2025

[2] https://developers.google.com/search/docs/appearance/page-experience accessed: 29.12.2025

[3] https://web.dev/articles/optimize-cls accessed: 29.12.2025

[4] https://web.dev/articles/vitals-tools accessed: 29.12.2025

Metric	Good	Needs improvement	Poor
LCP	$\leq 2.5\,s$	$> 2.5\,s$ and $\leq 4.0\,s$	$> 4.0\,s$
INP	$\leq 200\,ms$	$> 200\,ms$ and $\leq 500\,ms$	$> 500\,ms$
CLS	≤ 0.10	> 0.10 and ≤ 0.25	> 0.25

Table 2.1: Core Web Vitals thresholds

2.1.1 Largest Contentful Paint

LCP measures loading performance by reporting the render time of the largest content element visible within the viewport (commonly, a large image, video, or block of text). Because it approximates when the main content becomes visible to users, LCP is often used as a proxy for perceived load completion.[5]

In practice, LCP is influenced by multiple steps on the critical rendering path, including server response behavior, render-blocking resources, image loading, and client-side rendering. Rendering strategy can affect LCP because strategies that deliver meaningful HTML content earlier (e.g., Server Side Rendering (SSR) or SSG) can make the largest content element discoverable and prioritizable earlier in the loading process, whereas heavy Client Side Rendering (CSR) pages may delay identification and painting of the largest content element until after JavaScript execution.[6]

2.1.2 Interaction to Next Paint

INP measures responsiveness by capturing the latency of user interactions, specifically the delay between a user action (e.g., click, tap, keypress) and the next paint after the event is processed. Therefore, INP reflects how quickly a page responds noticeably to user input. INP is affected by factors such as main thread blocking, JavaScript execution time, and event handler efficiency.[7]

In 2024, Google introduced INP as a replacement for First Input Delay (FID), which only measured the delay of the first user interaction. This reflects a shift to capturing the responsiveness across interactions during the entire lifespan of a page, rather than just the initial interaction.[8] In practice, INP is often degraded by long tasks on the main thread, expensive JavaScript operations (e.g., parsing or rendering), and synchronous event handlers that block the main thread. Typical optimizations focus on breaking up long tasks, deferring non-critical work, and reducing main-thread contention.[9]

[5] https://web.dev/articles/lcp accessed: 29.12.2025
[6] https://web.dev/articles/optimize-lcp accessed: 29.12.2025
[7] https://web.dev/articles/inp accessed: 29.12.2025
[8] https://web.dev/blog/inp-cwv-launch accessed: 29.12.2025
[9] https://web.dev/articles/optimize-inp accessed: 29.12.2025

2.1.3 Cumulative Layout Shift

CLS measures visual stability by quantifying unexpected layout shifts that occur during the lifespan of a page. Such shifts can be disruptive to users, causing misclicks and creating a perception of instability. It is a unitless score computed from observed shifts in the layout. It is calculated as the product of an impact fraction and a distance fraction.[10]

Common causes of poor CLS include media without reserved dimensions, dynamically injected content that pushes existing elements and font swaps that change text dimensions after initial rendering. Mitigations therefore focus on reserving layout space, keeping placeholders stable, and controlling dynamic loading resources.[11]

2.2 Rendering Strategies

In the context of this thesis, rendering describes the process of producing HTML (or an equivalent Document Object Model (DOM) structure) from application code and data, which the browser then parses to construct the DOM and display the interface. Modern frameworks support multiple strategies that differ in where and when the HTML generation occurs, in the browser after JavaScript execution, on the server before sent to the client, ahead of time during build, or a combination thereof. Each strategy has trade-offs in terms of perceived performance, cacheability, SEO, and operational complexity.[12]

From a performance perspective, the rendering strategy influences both the speed at which meaningful content is delivered to the user (affecting LCP) and the efficiency of interactivity handling (affecting INP). Approaches that deliver prerendered HTML can improve LCP by allowing browsers to start rendering content sooner, while strategies that minimize main-thread blocking and optimize JavaScript execution can enhance INP by ensuring responsiveness to user interactions. Many systems therefore adopt a hybrid approach, selecting the strategy per route (e.g., static generation for marketing pages, server-side rendering for user dashboards) to balance performance, scalability, and complexity.[13]

Table 2.2 summarizes the rendering strategies and their trade-offs.

[10] https://web.dev/articles/cls accessed: 29.12.2025
[11] https://web.dev/articles/optimize-cls accessed: 29.12.2025
[12] https://web.dev/articles/rendering-on-the-web accessed: 30.12.2025
[13] https://nextjs.org/learn/pages-router/data-fetching-two-forms accessed: 30.12.2025

Strategy	HTML produced	Advantages	Limitations
CSR	In the browser	High interactivity after initial load; straightforward content delivery via Content Delivery Network (CDN).	Higher client-side JavaScript and CPU cost; delayed initial content visibility; potential SEO issues.
SSR	On the server per request	Early content delivery; supports request-time freshness and personalization.	Latency variability; runtime server overhead; hydration increases client work.
SSG	At build time	Excellent cacheability and predictable delivery via CDN; minimal runtime infrastructure.	Build time grows with page volume; updates require rebuild and deploy; hydration mismatches; limited request-time personalization.

Table 2.2: Condensed comparison of rendering strategies and their trade-offs

2.2.1 Client-Side Rendering

CSR generates the page content entirely in the browser using JavaScript. A typical CSR workflow involves the server delivering a minimal HTML shell along with JavaScript bundles. Once the JavaScript is downloaded and executed, it fetches data and constructs the DOM dynamically.[14] This model is commonly associated with Single Page Application (SPA), where navigation between views occurs client-side without full page reloads.

Benefits

CSR can provide a highly interactive, application-like experience once the initial bundles are loaded, because navigation may reuse previously loaded code and state. It also simplifies some deployment models, since the origin can often serve static assets via a CDN, while application behavior is implemented in the browser.

Limitations

With CSR, significant work is deferred to the client device. As applications grow, the size of JavaScript bundles can increase, leading to longer download and parse times. This additional initial work on the client can negatively impact LCP, especially on slower devices or networks.[15] In addition, SEO can be challenging with CSR, since search engine crawlers may not fully execute JavaScript, leading to incomplete indexing of content. Furthermore, purely client-side routing and error handling may lead to incorrect HTTP status codes being returned for certain

[14] https://developer.mozilla.org/en-US/docs/Glossary/CSR accessed: 20.12.2025
[15] https://web.dev/articles/rendering-on-the-web accessed: 30.12.2025

pages, which can further impact SEO.[16]

2.2.2 Server-Side Rendering

SSR generates HTML on the server for each request and sends the fully rendered HTML to the client. In modern frameworks, this is often combined with hydration, where the server-rendered HTML is enhanced with client-side JavaScript to enable interactivity.[17]

Benefits

SSR can significantly improve LCP by delivering fully formed HTML to the browser, allowing it to render meaningful content immediately. This is particularly beneficial for SEO, as search engines can easily index the prerendered content.

Limitations

However, SSR introduces infrastructure complexity, as servers must handle rendering logic and scale to meet request loads. This can lead to variable latency, as rendering times depend on server load and request complexity.[18] Additionally, hydration can add to the client-side workload, potentially impacting INP if not optimized properly.[19] If pages are hydrated, avoiding hydration mismatches (server/build HTML vs. client render) can be challenging.

2.2.3 Static Site Generation

SSG prerenders HTML pages at build time, producing static HTML files that can be directly served. A defining property of SSG is that, for a given URL, users always receive the same prerendered HTML until the site is rebuilt.[20]

Benefits

SSG typically offers fast, predictable delivery because requests are served from caches and CDNs without request-time rendering. This model scales well and can reduce runtime attack surface, since application logic does not need to execute for each request.

[16] https://developers.google.com/search/docs/crawling-indexing/javascript/javascript-seo-basics accessed: 30.12.2025

[17] https://developer.mozilla.org/en-US/docs/Glossary/SSR accessed: 30.12.2025

[18] https://web.dev/articles/rendering-on-the-web accessed: 30.12.2025

[19] https://react.dev/reference/react-dom/client/hydrateRoot accessed: 30.12.2025

[20] https://developer.mozilla.org/en-US/docs/Glossary/SSG accessed: 30.12.2025

Limitations

The main limitation is freshness: content updates require rebuilding and redeploying the site. For large sites, build time can become a bottleneck as the number of pages grows. When personalization is needed, developers often reintroduce CSR or selectively apply SSR, which reintroduces some runtime costs. If pages are hydrated, avoiding hydration mismatches (server/build HTML vs. client render) can be challenging.

2.3 Search Engine Optimization

SEO refers to the set of technical and structural measures that improve the visibility and discoverability of web content in search engine results pages.[21] From a technical perspective, SEO is primarily concerned with how search engines crawl, interpret, and index web documents, and how reliably these documents can be associated with relevant user queries.[22]

As discussed in Section 2.1, search engines incorporate user experience signals into ranking considerations. In particular, CWV provide standardized field metrics for loading performance, responsiveness, and visual stability. Poor values for metrics such as LCP, INP, and CLS can therefore affect not only perceived usability, but also the competitiveness of a page in search results as part of broader page experience evaluation. Consequently, technical SEO and performance engineering are closely coupled: architectural choices that reduce critical-path work and improve delivery behavior often contribute directly to better CWV outcomes.

Modern search engines operate largely on an HTML-first indexing model. While they increasingly support client-side JavaScript execution, the reliability, completeness, and timing of content indexing still depend heavily on the availability of fully rendered HTML documents at crawl time.[23] Consequently, web architectures that expose content only after client-side execution or asynchronous data loading may introduce uncertainty in how and when content is indexed.

Several technical factors are commonly identified as beneficial for SEO. These include fast initial load times,[24] stable, meaningful and canonical URLs, consistent internal linking structures, and the availability of descriptive metadata such

[21] https://developers.google.com/search/docs/fundamentals/seo-starter-guide accessed: 07.02.2026

[22] https://developers.google.com/search/docs/fundamentals/how-search-works accessed: 07.02.2026

[23] https://web.dev/articles/rendering-on-the-web accessed: 07.02.2026

[24] https://developers.google.com/search/docs/appearance/page-experience accessed: 07.02.2026

as document titles and language annotations.[25]

In multilingual web applications, SEO further depends on the clear separation of language variants through distinct URLs and explicit signaling of language relationships between pages.[26]

Importantly, SEO is not limited to marketing considerations, but represents a technical quality attribute of web systems. Architectural decisions related to rendering strategies, content delivery mechanisms, and localization directly influence how effectively search engines can discover and index content. As a result, SEO considerations are increasingly addressed at the architectural level rather than being treated as a post-hoc optimization.

At the same time, this thesis intentionally focuses on the technical dimensions of SEO (crawlability, indexability, delivery performance, and localization signaling). Content strategy and editorial quality are not addressed, even though the most important prerequisite for sustained discoverability is typically relevant, well-structured, and user-oriented content. In other words, technical SEO can improve accessibility and reliability of indexing, but it cannot compensate for missing or poorly organized information content.

2.3.1 Machine Discoverability and AI Consumption

While SEO has traditionally focused on visibility in search engine results for human users, its underlying principles are increasingly relevant for automated systems and Artificial Intelligence (AI)-driven content consumers. Large language models, retrieval-based assistants, and search-integrated AI features rely on web-crawled and indexed content as an important knowledge source (Baack, 2024). Like search engines, these systems benefit from structured, accessible, and language-explicit content representations. Fully rendered HTML documents, stable URLs, and consistent internal linking support reliable crawling and interpretation, whereas content that depends heavily on client-side execution may be less accessible or less consistently interpreted.

This also applies to Google's *AI Overviews* and *AI Mode*. Google states that-Explicit indication of fallback-driven language change no additional technical requirements or special optimizations are needed beyond established SEO best practices. Instead, the same foundations remain relevant: crawlability, indexability, accessible textual content, internal linking, and a good page experience.[27]

[25] https://developers.google.com/search/docs/specialty/international/managing-multi-regional-sites accessed: 07.02.2026

[26] https://developers.google.com/search/docs/specialty/international/localized-versions accessed: 07.02.2026

[27] https://developers.google.com/search/docs/appearance/ai-features accessed: 07.02.2026

In multilingual contexts, the availability of language-specific content further affects how AI systems associate information with linguistic and regional contexts. Distinct, localized URLs and explicit language signaling contribute to clearer semantic boundaries, reducing ambiguity in automated content consumption. As AI systems increasingly mediate access to information, discoverability becomes a shared concern for both search engines and AI-driven applications.

Consequently, SEO can be understood not only as a mechanism for ranking optimization, but as a broader technical discipline for ensuring machine-readable accessibility of web content. Architectural decisions that improve crawlability, determinism, and localization transparency therefore support both conventional search engines and emerging AI-mediated discovery systems.

2.4 Weblate

Weblate[28] is an open-source, web-based localization platform that supports continuous localization by integrating directly with a version control repository. Instead of keeping translations in a separate tool-specific silo, Weblate synchronizes translation files with the upstream repository: it pulls changes from the repository, provides a browser-based interface for translation work, and can push approved translations back to the repository.[29] This makes translation contributions accessible to non-developers while keeping the source of truth in the same workflow used for software development.

In this thesis, Weblate is used to translate a *React* application that relies on *FormatJS* message formatting. The application's message catalogs are stored in *GitLab*. Weblate can be configured to track these files as a translation component and to contribute changes back through the Git workflow. Additionally, synchronization steps can be automated in GitLab continuous integration pipelines, for example by triggering repository updates or invoking Weblate's REST Application Programming Interface (API) to keep translation content consistent with the current code base and add new components and messages as they are introduced during development. Furthermore, the current translation status is managed in Weblate, allowing the system to pull a config through the API to provide information about translation completeness and verification to the user.[30]

[28] https://weblate.org accessed: 30.12.2025
[29] https://docs.weblate.org/en/latest/devel/integration.html accessed: 30.12.2025
[30] https://docs.weblate.org/en/latest/api/ accessed: 30.12.2025

3 Requirements

This chapter specifies the requirements for the solution implemented in this thesis: static site generation for eligible routes and a route-aligned, namespaced internationalization architecture integrated with a continuous translation workflow. The requirements are derived from the baseline system constraints and the goals introduced in Chapter 1. Each requirement is stated in a way that allows verification and is accompanied by an explicit validation method.

3.1 Validation Methods

Requirements are validated using one of the following methods:

- **Measurement:** quantitative comparison using build logs, timings, artifact sizes, or network traces.
- **Test:** automated verification via unit/integration tests, CI jobs, or reproducible scripts.
- **Inspection:** manual or scripted inspection of build artifacts, manifests, configuration, and output structure.

3.2 Context, Constraints, and Scope

3.2.1 Problem Context

The baseline system exhibits two main practical issues. Crawler-based prerendering increases fragility and cost as route/language coverage grows, and monolithic translation catalogs couple runtime payload and parsing cost to the total size of the application rather than to the active route, while lacking the ability to support incremental rollout of translations without mixed-language User Interface (UI).

3.2.2 Scope

The requirements target improvements in:

- **Static generation tooling** for eligible routes, executed in Node.js.
- **Route-aligned localization** via namespaced translation resources and selective loading.
- **Translation workflow integration** with status metadata, quality states, governed fallback, and user disclosure.
- **CI automation** to prevent regressions and keep translation updates auditable.

3.2.3 Out of Scope

Backend delivery infrastructure, CDN strategy, and request-time personalization are out of scope. The target remains a hybrid model: only eligible routes are statically generated while other routes remain client-side rendered.

3.2.4 Definitions

The following definitions are used throughout the requirements to ensure precision and consistency.

Route. A *route* denotes a uniquely addressable application view that is associated with a canonical URL path and can be rendered independently.

Eligible Route. An *eligible route* is a route that is explicitly marked for static site generation in a build-time configuration. Only eligible routes may be statically generated.

Static HTML. A *static HTML* file is a prerendered HTML document emitted at build time at a deterministic location in the build artifact structure and intended to be served without server-side rendering at request time.

Language variant. A *language variant* of a route is a language-specific version of the route that is rendered with localized content according to the translation state of the required namespaces for that route.

Namespace. A *namespace* is a logical grouping of localized message identifiers. Namespaces may be shared across routes or be specific to individual routes.

Required Namespaces. For a given route r, the set of requiredNamespaces(r) denotes all namespaces whose messages are required to fully render the route.

Translation State. For each (l, n) with language l and namespace n, the *translation state* $\text{state}(l, n)$ is one of the following discrete values:

- **missing**: no translation is available for the namespace in the given language,
- **translated**: a translation exists but has not been approved,
- **approved**: a translation exists and has been reviewed and approved.

Derived Predicates. We define the following predicates on translation states:

$$\text{hasTranslation}(l, n) \triangleq \text{state}(l, n) \in \{\textit{translated}, \textit{approved}\},$$

$$\text{isApproved}(l, n) \triangleq \text{state}(l, n) = \textit{approved}.$$

Route Availability. A route r is considered *available* (and thus publishable) in language l if and only if

$$\forall n \in \text{requiredNamespaces}(r) : \text{hasTranslation}(l, n).$$

Default Language. The *default language* is the primary fallback language of the application and is guaranteed to be available for all routes, i.e.

$$\forall r : \ \forall n \in \text{requiredNamespaces}(r) : \text{hasTranslation}(\textit{defaultLanguage}, n).$$

SSG Run. An *SSG run* is a single build-time execution of the static site generation process. Given the build configuration (including the set of eligible routes), the set of supported languages, and the translation status information, an SSG run:

1. determines the set of *available* language variants (r, l) of eligible routes,
2. renders each available (r, l) to produce static HTML files in a deterministic, non-browser rendering environment, and
3. emits the resulting static HTML files at deterministic locations in the build output structure.

An SSG run may be invoked locally or as part of a CI workflow.

Build. The *build* denotes the complete build and bundling process of the React frontend project. It transforms the project source code and configuration (including route eligibility, language settings, and translation inputs) into a deployable build artifact structure containing the compiled client-side bundles (e.g., JavaScript and CSS), resolved static assets, and where applicable, the static HTML files emitted by a SSG run. The build may be executed locally or within a CI workflow.

Runtime. The *runtime* refers to the environment in which the application executes after being built and deployed. It includes the browser or other execution context where the static HTML files are served and interacted with by users.

Translation File Schema

The localization resources consumed by the system shall be stored as line-oriented text files with a deterministic filename and record format.

Filename schema. A translation bundle file for namespace n and language l shall use the filename

`<namespace-slug>.<language>.txt`

where `<namespace-slug>` is the namespace identifier in slug-case and `<language>` is the language code of the bundle.

Content schema. A translation bundle file shall consist of one or more message entries, each encoded on exactly one line in the form

`<namespaceInCamelCase>[.<identifier>]*=<value>`

where the key of the key-value pair can consist of multiple segments separated by dots. The file content shall end with a newline character.

3.3 Specification Conventions

To keep the requirements precise and easy to review, this chapter follows two conventions.

Requirement phrasing. Each statement is written as a single, complete sentence using normative language (*shall* for mandatory behavior, *should* for desired behavior). For the structure of these sentences, the requirements follow the SOPHIST *MASTeR* templates: functional requirements use the *FunctionalMASTER* pattern and non-functional requirements use the non-functional templates (in particular the *PropertyMASTER* for measurable properties and comparisons).[1]

Structuring for non-functional requirements. Non-functional requirements are organized according to the product-quality characteristics of ISO/IEC 25010. This provides a stable taxonomy for quality goals and helps to avoid mixing unrelated concerns in one subsection.[2]

[1] https://sophist.de/w4f/ accessed: 25.03.2026

[2] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 accessed: 25.03.2026

3.4 Functional Requirements

Functional requirements define the expected behavior of the system in terms of the results a product or process shall produce and the functions that a system or system component shall perform. ISO/IEC/IEEE 24765 characterizes functional requirements as statements identifying required results and as requirements specifying system functions ('ISO/IEC/IEEE 24765', 2017).

The functional requirements are grouped by the main solution components: the static site generation, namespaced localization, incremental translation logic, and translation lifecycle integration with CI governance.

3.4.1 Static Site Generation

FR-SSG-01 (Configurable hybrid rendering). The system shall generate static HTML files at build time for all routes marked as eligible for SSG in the build configuration. For routes not marked as eligible, a SSG run shall not emit route-specific static HTML files, and those routes shall be rendered client-side at runtime.

Validation (Inspection): Given a build configuration with a defined set of eligible and non-eligible routes, execute an SSG run and verify that static HTML files are emitted only for eligible routes, while non-eligible routes do not have corresponding static HTML files in the output.

FR-SSG-02 (Language variants for eligible routes). For each route eligible for static generation, an SSG run shall emit language-specific static HTML files for all languages in which the route is available, i.e. for which all required namespaces satisfy $\text{state}(l, n) \in \{\textit{translated}, \textit{approved}\}$.

Validation (Inspection): Given an eligible route and a set of languages with varying translation states across required namespaces, execute the build and verify that static HTML files are emitted for languages in which all required namespaces have the required translation state.

FR-SSG-03 (Deterministic server-side rendering environment). During SSG, the system shall render routes in a deterministic, non-browser environment (Node.js) that provides functional equivalents for the browser APIs required by route rendering, including DOM access and storage APIs.

Validation (Inspection): Execute SSG for routes that access DOM and storage APIs and verify that rendering completes successfully and produces correct static HTML output without runtime errors or network access.

FR-SSG-04 (Module and asset resolution support). The SSG shall support the project's modules and asset resolution strategy required for route ren-

dering, such that static HTML outputs correctly reference build-emitted assets.

Validation (Inspection): Run the SSG end-to-end and verify that eligible routes render successfully and that all assets referenced in the generated static HTML correspond to existing build artifacts.

FR-SSG-05 (Render isolation). The SSG shall isolate individual route renders such that timers, global state, and other rendering side effects created during one route's render do not influence subsequent route renders. Rendering side effects shall not persist beyond the lifetime of the SSG process.

Validation (Inspection): Execute static generation on routes that intentionally introduce timers or global state and verify that subsequent route renders are unaffected and that the generation process terminates with no outstanding timers or persistent side effects.

FR-SSG-06 (Incremental regeneration). The SSG should support incremental regeneration of static HTML files, such that only a subset of routes (e.g., those with changed code or changed translation state) are re-rendered when the is executed with an incremental flag.

Validation (Inspection): Run the SSG with a set of routes, then make a change to one route's code or translation state and run the SSG again with the incremental flag and verify that only the changed route is re-rendered and that unchanged routes are not re-rendered.

FR-SSG-07 (Error-free hydration of statically generated pages). For every statically generated route-language variant (r, l) emitted by an SSG run, the system shall hydrate the corresponding page on the client at runtime without hydration errors or hydration mismatch warnings.

Validation (Inspection): Serve the static artifacts produced by an SSG run and load each generated page in a browser with developer tools open. Inspect the browser console during initial load and verify that no hydration mismatch warnings or hydration-related errors are reported, and inspect that the page becomes interactive after load.

3.4.2 Namespaced Localization and Selective Loading

FR-I18N-01 (Namespaced message bundles). Localized messages shall be organized into namespaces. The system shall define a shared baseline namespace that is required by all routes and one or more route-specific namespaces used only by routes that declare them as required.

Validation (Test): Verify that for each language and each namespace, a corresponding message bundle artifact exists and conforms to the expected structure

(i.e., location and format). Verify that the routing configuration declares the baseline namespace for all routes.

FR-I18N-02 (Route-driven namespace resolution activation). The runtime shall determine the set of required namespaces for the active route from the route's metadata and shall ensure that all required namespaces for the active route are loaded and activated before rendering localized route content.

Validation (Inspection): For a representative set of routes with differing required namespaces, navigate between routes and assert that the resolved namespaces equal the route's declared metadata and localized content for that route is rendered using those namespaces.

FR-I18N-03 (Caching and deduplication). The message loader shall cache successfully loaded namespace bundles per (*language*, *namespace*) pair and shall deduplicate concurrent load requests for the same (*language*, *namespace*) pair such that at most one load operation is performed for each unique (*language*, *namespace*) pair during a session.

Validation (Inspection): Trigger multiple concurrent requests for the same (*language*, *namespace*) pair and verify that only one underlying load operation is performed. Verify that subsequent requests for the same (*language*, *namespace*) pair during the same session are served from cache without additional load operations.

FR-I18N-04 (SSG-compatible message loading). During SSG, message loading shall be deterministic and shall not depend on network access. The system shall load required namespace bundles from build artifacts available in the local filesystem.

Validation (Test): Run SSG without network access and verify that localized static HTML files are generated correctly for eligible routes and contain localized content derived from build artifacts.

FR-I18N-05 (Bundle integrity and schema validation). Namespace translation files shall conform to the defined translation file schema and shall be validated at build time to prevent runtime failures due to malformed localization files.

Validation (Test): Introduce a malformed translation file and verify that the build job fails.

FR-I18N-06 (Preload for initial render). On initial page load, the runtime should preload the baseline namespace and all route-required namespaces for the initial route to prevent visible flicker or late text replacement.

Validation (Inspection): Perform initial page load and verify that all required namespaces for the initial route are loaded before the first render.

FR-I18N-07 (Prefetch strategy). The runtime should implement a prefetch strategy to proactively load namespaces for routes that the user is likely to navigate to next, based on heuristics such as link visibility or user interaction patterns.

Validation (Inspection): Simulate user navigation patterns and verify that for likely next routes the translation resources are prefetched before navigation occurs.

FR-I18N-08 (Data-saving option). The system should provide a data-saving mode that limits namespace loading to only the baseline namespace and the currently active route's required namespaces, deferring prefetching and non-essential loads.

Validation (Inspection): Enable the data-saving mode and verify that only the baseline and active route namespaces are loaded, while prefetching is disabled.

3.4.3 Incremental Translation Strategy

FR-INC-01 (Incremental rollout without mixed-language UI). The system shall support partially translated languages while ensuring route-level language consistency. A route r shall be considered available (and publishable) in language l if and only if for all namespaces $n \in \text{requiredNamespaces}(r)$, the translation state satisfies

$$\text{state}(l, n) \in \{\textit{translated}, \textit{approved}\}.$$

If a route is not available in language l, it shall not be published for language l.

Validation (Inspection): Given a route requiring namespaces $\{n_1, n_2\}$ and a language l with $\text{state}(l, n_1) = \textit{translated}$ and $\text{state}(l, n_2) = \textit{missing}$, verify that the route has no published output for language l and that navigation does not produce a language-specific URL for (r, l).

FR-INC-02 (Deterministic navigation under missing translations). If a target route is not available in the currently active language, navigation elements shall resolve links to the corresponding route in the default language rather than generating a language-specific URL for an unavailable route.

Validation (Inspection): From a page in a non-default language, render navigation to a target route that is unavailable in that language. Verify that the generated link points to the default-language URL for the target route and does not point to a non-existent language-specific URL.

FR-INC-03 (Explicit indication of fallback-driven language change). If a navigation element points to the default-language version of a route because that route is unavailable in the currently active language, the UI shall indicate that activating the element will open the destination in the default language.

Validation (Inspection): Inspect a navigation element on a page in a non-default active language l. If the target route is unavailable in l and therefore links to the default-language version, verify that the UI clearly indicates before navigation that activating the element will open the destination in the default language.

FR-INC-04 (Route-specific language eligibility in language selection). The language selector shall offer a language option for the active route only if the route is available in that language (i.e., all required namespaces satisfy state$(l, n) \in \{$*translated*, *approved*$\}$ for the route).

Validation (Inspection): On a route with required namespaces n_1, n_2, set language l such that at least one required namespace has state$(l, n) =$ *missing*. Verify that language l is not offered in the language selector for that route, while it may remain offered on routes that are available in l.

FR-INC-05 (Synchronized language selector). The language selector shall update on navigation such that the set of offered languages always reflects the availability of the newly active route and does not display stale language options from the previous route.

Validation (Inspection): Navigate across routes with different availability sets and verify that the language selector updates immediately after navigation such that only languages available for the current route are offered.

FR-INC-06 (Deterministic behavior for direct access for unavailable language URLs). If a user directly requests a language-specific URL for a route that is unavailable in that language, the system shall respond deterministically by redirecting to the default-language URL.

Validation (Inspection): Request an unavailable language-specific route URL. Verify that the response is a redirect to the default language URL.

FR-INC-07 (User disclosure for unapproved translations). If a route is published in the currently active language and at least one namespace required by the route has translation state **translated** (and not **approved**) for that language, the UI shall display a clear disclosure indicating that the page contains translations that have not yet been reviewed/approved.

Validation (Inspection): Configure a route whose required namespaces include at least one namespace with state$(l, n) =$ *translated* and none with *missing*, load the route in language l, and verify that the disclosure is shown. Repeat with all required namespaces in state *approved* and verify that the disclosure is not shown.

FR-INC-08 (Sitemap completeness). On production releases, the CI-pipeline shall generate a sitemap that includes *all and only* published language-specific routes.

Validation (Inspection): Simulate a production release with a known set of published and non-published route-language variants. Inspect the generated sitemap and verify that all published route-language URLs are included and that no non-published route-language URLs are included.

FR-INC-09 (`lastmod` for published pages). For every published URL included in the sitemap, the corresponding `<url>` entry shall include a `<lastmod>` value that reflects the last modification time of the published page.

Validation (Inspection): Simulate a production release with a known set of published routes. Inspect the generated sitemap and verify that each corresponding `<url>` entry contains a `<lastmod>` element.

FR-INC-10 (IndexNow submission for changed pages). After a successful production release, the CI-pipeline shall submit an *IndexNow* request to trigger re-indexing for the set of published route-language URLs whose emitted static HTML output changed compared to the previous production release.

Validation (Inspection): Perform a production release where only a known subset of published routes changes. Inspect the pipeline logs and/or captured network requests and verify that an IndexNow submission is performed and that the submitted URL set corresponds to exactly the changed published route-language URLs, using the configured IndexNow authentication parameters.

3.4.4 Translation Lifecycle Integration and CI Governance

FR-LC-01 (Translation status artifact consumption). The runtime shall consume a translation status artifact that defines the translation state for each (*language*, *namespace*) pair. The translation state shall be one of **missing**, **translated**, **approved** and shall be used by runtime decision logic for route availability, navigation fallback, and language selection.

Validation (Inspection): Verify that a translation status artifact exists, that it defines the translation state for each relevant (*language*, *namespace*) pair, and that the runtime logic for route availability and navigation behaves according to the defined translation states.

FR-LC-02 (Check validity of translation status artifact). The CI pipeline shall validate that the translation status artifact is consistent with the set of namespace message bundles produced by the build. In particular, for every (*language*, *namespace*) entry referenced in the translation status artifact, the corresponding namespace bundle shall exist in the build artifacts. If any referenced bundle is missing, CI shall fail and report the affected language(s) and namespace(s).

Validation (Test): Run the validation job with a deliberately missing namespace

bundle referenced by the status artifact and verify that the CI pipeline fails with language/namespace diagnostics. Verify that the job succeeds when all referenced bundles are present.

FR-LC-03 (Default-language change propagation for existing translations). The system shall provide a developer-invoked translation update task that detects uncommitted changes to default-language message texts relative to `HEAD` and, for every affected (*language*, *namespace*) translation file in which the changed identifier exists, updates the corresponding translated value when executed in a machine-translation mode. Updated entries shall be explicitly marked as machine-updated and requiring human review.

Validation (Inspection): Modify the defaultMessage of an existing identifier in the default language, regenerate the default-language template, and execute the translation update task in machine-translation mode. Verify that the corresponding entry is updated in each affected non-default translation file where the identifier exists, and that the entry is annotated as requiring review.

FR-LC-04 (Machine-translation fill for newly introduced identifiers). When executed in machine-translation mode, the developer translation update task shall detect identifiers that exist in the default-language template for a given namespace but are missing in a corresponding non-default translation file, and shall add entries for those identifiers using machine translation derived from the default-language `defaultMessage`. Machine-inserted entries shall be explicitly marked as machine-generated and requiring human review.

Validation (Inspection): Introduce a new message identifier, execute the translation update task in machine-translation mode, and verify that the missing entry is added to the affected translation file(s) with a machine-translated value and an explicit review-required annotation.

FR-LC-05 (View for unreviewed translations). The system shall provide a dedicated visualization for reviewers to inspect and review translations that have not yet been approved. This view shall provide visible indicators to quickly identify unapproved translations.

Validation (Inspection): Access the unapproved translations view and verify that all translations with **translated** state have a clear visual indicator of their unapproved status.

FR-LC-06 (Context visibility for unapproved translations). The translation workflow shall provide a medium to view unapproved translations within their visual context to assist the reviewer in the approval process.

Validation (Inspection): For a route with unapproved translations, access the contextual review medium and verify that unapproved translations are displayed in their visual context to assist in the review process.

FR-CI-01 (Message identifier convention enforcement). CI shall enforce message identifier conventions to ensure stable keys and tool compatibility. On violations, CI shall fail and produce actionable diagnostics identifying the offending identifiers and their source locations. Message identifiers shall follow the convention `[namespace].[key]` where `namespace` corresponds to the declared namespace and `key` is a stable, descriptive identifier for the message.

Validation (Test): Introduce identifiers violating the convention and verify that the CI job fails and reports the identifier, the file (or extraction location), and the violated rule.

FR-CI-02 (Namespace dependency and boundary enforcement). CI shall ensure that routes only use message identifiers from namespaces they declare as required. Any usage of an identifier belonging to an undeclared namespace shall cause CI to fail to prevent missing localization dependencies at runtime.

Validation (Test): Create a route that references an identifier from an undeclared namespace. Verify that CI fails and reports the route, the identifier, the inferred namespace, and the missing namespace declaration.

FR-CI-03 (Auditable and deterministic translation synchronization). Translation template synchronization and the application of incoming translation updates shall be automated in CI and shall produce deterministic, reviewable changes. The resulting diffs shall be reproducible from the same inputs and shall not include unrelated or non-deterministic formatting changes.

Validation (Inspection): Run the synchronization/update pipeline twice from the same repository state and translation inputs and verify that the produced diffs are identical. Verify that changes are limited to translation artifacts and associated metadata.

FR-CI-04 (Structured Merge Requests). Translation-related CI jobs shall produce scoped and manageable merge requests according to a defined partitioning policy (e.g., per language, per namespace, or per logical change set). The pipeline shall prevent the creation of large monolithic merge requests when the changes can be partitioned according to the policy.

Validation (Inspection): Trigger translation updates affecting multiple languages and namespaces and verify that merge requests are created according to the partitioning policy. Verify that each merge request contains only the intended subset of changes and includes machine-generated context (e.g., languages/namespaces affected).

An overview of the functional requirements is provided in Table 3.1.

ID	Short Title	Shall/ Should	Validation Method
FR-SSG-01	Configurable hybrid rendering	shall	Inspection
FR-SSG-02	Language variants for eligible routes	shall	Inspection
FR-SSG-03	Deterministic server-side rendering environment	shall	Inspection
FR-SSG-04	Module and asset resolution support	shall	Inspection
FR-SSG-05	Render isolation	shall	Inspection
FR-SSG-06	Incremental regeneration	should	Inspection
FR-SSG-07	Error-free hydration of statically generated pages	shall	Inspection
FR-I18N-01	Namespaced message bundles	shall	Test
FR-I18N-02	Route-driven namespace resolution activation	shall	Inspection
FR-I18N-03	Caching and deduplication	shall	Inspection
FR-I18N-04	SSG-compatible message loading	shall	Test
FR-I18N-05	Bundle integrity and schema validation	shall	Test
FR-I18N-06	Preload for initial render	should	Inspection
FR-I18N-07	Prefetch strategy	should	Inspection
FR-I18N-08	Data-saving option	should	Inspection
FR-INC-01	Incremental rollout without mixed-language UI	shall	Inspection
FR-INC-02	Deterministic navigation under missing translations	shall	Inspection
FR-INC-03	Explicit indication of fallback-driven language change	shall	Inspection
FR-INC-04	Route-specific language eligibility in language selection	shall	Inspection
FR-INC-05	Synchronized language selector	shall	Inspection
FR-INC-06	Deterministic behavior for direct access for unavailable language URLs	shall	Inspection
FR-INC-07	User disclosure for unapproved translations	shall	Inspection
FR-INC-08	Sitemap completeness	shall	Inspection
FR-INC-09	`lastmod` for published pages	shall	Inspection
FR-INC-10	IndexNow submission for changed pages	shall	Inspection
FR-LC-01	Translation status artifact consumption	shall	Inspection
FR-LC-02	Check validity of translation status artifact	shall	Test
FR-LC-03	Default-language change propagation for existing translations	shall	Inspection
FR-LC-04	Machine-translation fill for newly introduced identifiers	shall	Inspection
FR-LC-05	View for unreviewed translations	shall	Inspection
FR-LC-06	Context visibility for unapproved translations	shall	Inspection
FR-CI-01	Message identifier convention enforcement	shall	Test
FR-CI-02	Namespace dependency and boundary enforcement	shall	Test
FR-CI-03	Auditable and deterministic translation synchronization	shall	Inspection
FR-CI-04	Structured Merge Requests	shall	Inspection

Table 3.1: Overview of functional requirements

3.5 Non-Functional Requirements

In this thesis, *non-functional requirements* describe the qualities and constraints that govern how the software operates, rather than the concrete functionality it provides. According to ISO/IEC/IEEE 24765, a non-functional requirement specifies not what the software will do, but how it will do it.('ISO/IEC/IEEE 24765', 2017)

To make the quality goals easier to locate and to keep the chapter consistent with the ISO/IEC 25010 taxonomy, the non-functional requirements are grouped by ISO/IEC 25010 characteristics. The structure does not imply that the remaining characteristics are irrelevant, but it reflects the set that is directly addressed by this thesis.

3.5.1 Performance Efficiency

This characteristic captures time behavior and resource utilization. In the context of this thesis, it covers both build-time performance of SSG and runtime efficiency of translation loading.

Benchmark protocol (P1). Unless stated otherwise, performance measurements shall be conducted on the same build runner and operating system image, with identical route and language coverage, with a clean workspace, and across repeated runs. Results shall be reported at least as median and standard deviation.

NFR-01 (Reduced SSG build time). For identical route and language coverage, the implemented SSG shall achieve a lower prerendering duration than crawler-based prerendering when measured under Protocol P1. The median prerendering duration of the target approach shall be strictly lower than the baseline median.

Validation (Measurement): Run baseline and target builds with identical inputs under Protocol P1 and compare prerender duration distribution.

NFR-02 (Parallel scalability). The SSG shall support parallel route rendering. When increasing the worker count on a multi-core system, the median throughput shall not degrade, and it shall demonstrate at least one configuration whose median prerendering duration is strictly lower than the single-worker configuration under Protocol P1.

Validation (Measurement): Run SSG with multiple worker counts under Protocol P1 and report scaling curve and the best-performing configuration.

NFR-03 (Largest Contentful Paint on public pages). For eligible public-facing routes, the delivered page shall achieve a LCP of at most 2.5 s at the 75$^{\text{th}}$ percentile of page loads. Additionally, the LCP should be lower than the LCP achieved by the crawler-based prerendering baseline for the same route and language coverage under the same measurement setup.

Validation (Measurement): Measure LCP across repeated runs (or field samples) and report the 75$^{\text{th}}$ percentile and compare the target against the crawler-based baseline for identical routes/locales and identical test conditions.

NFR-04 (Route-bounded translation payload). For CSR routes, the runtime shall only download translation resources required by the active route. Specifically, for a given navigation state, the set of fetched namespaces shall be limited to shared baseline namespaces and the namespaces declared as required by the active route. No other namespaces shall be transferred unless an explicit prefetch mechanism is triggered.

Validation (Measurement): Record a network trace for first load and representative navigations and verify that requested namespaces equal the declared set and compare transferred bytes to a monolithic baseline.

3.5.2 Reliability

This characteristic covers predictable behavior over time and under fault conditions. For this thesis, the focus is on deterministic outputs and robust behavior when translation resources are incomplete or temporarily unavailable.

NFR-05 (Deterministic artifacts). Given identical inputs (code revision, configuration, translation bundles, lockfiles, and build settings), generated HTML output for eligible routes shall be deterministic for local and CI environments. Output files shall be byte-identical across repeated runs.

Validation (Inspection): Run SSG twice in a clean environment with identical inputs and compare emitted files and fail on differences outside the whitelist.

NFR-06 (Consistent behavior between SSG and CSR). Localization behavior (namespace resolution, and quality-state handling) shall yield the same user-visible outcome for a route delivered via SSG and the same route delivered via CSR, for the same language and the same translation quality-state conditions.

Validation (Inspection): For a subset of eligible routes compare rendered primary content and quality-state indicators between SSG and CSR delivery.

NFR-07 (Resilient failure handling). If loading a required namespace fails, the system shall not crash navigation and shall not enter a blank-screen or unhandled-error state. The system shall transition into a recoverable state that

prevents rendering mixed-language or partially resolved content without disclosure, and enables recovery by redirecting to a defined fallback language.

Validation (Inspection): Inject a fault into namespace fetching and verify no uncaught errors, continued navigability, and correct recoverable-state behavior.

3.5.3 Maintainability

This characteristic captures modifiability and analyzability. In the scope of this thesis, maintainability is primarily addressed through explicit contracts (route-to-namespace declarations) and avoiding brittle, unmaintained build dependencies.

NFR-08 (Route-to-namespace maintainability). Namespace dependencies shall be declared at the routing boundary in a single, reviewable location (the route configuration). The build and/or CI shall detect and fail if a route uses translation keys outside its declared namespace set.

Validation (Inspection + Test): Inspect the route configuration and verify the explicit namespace declarations. The CI pipeline shall fail if undeclared namespaces are used.

NFR-09 (Sustainable toolchain). Correctness of eligible-route SSG output shall not depend on crawling or browser-automation-based prerendering tools as a required step in the production pipeline.

Validation (Inspection): Document the production build pipeline and confirm that snapshot crawling is not on the critical path for generating eligible routes.

NFR-10 (Runtime telemetry for translation loading). The runtime should emit structured telemetry for translation loading, including: namespace load latency, cache hit/miss information, and namespace-load failures. Telemetry should include sufficient context to attribute events to a route and language.

Validation (Inspection + Test): Execute a controlled navigation scenario and verify that telemetry events are emitted with the required fields (route identifier, language, namespace, outcome, latency). Inject a failed namespace load and verify that failure telemetry is emitted.

NFR-11 (Developer documentation). Maintainability-relevant aspects of the implementation shall be documented in a reviewable and persistent form. This documentation shall include code comments for non-obvious implementation logic and wiki entries for architecture, build procedures, localization workflows, and maintenance-relevant operational knowledge.

Validation (Inspection): Inspect the source code and project wiki and verify that non-obvious implementation logic is explained by code comments and that

architecture, build procedures, and localization workflows are documented in the wiki.

3.5.4 Functional Suitability

This characteristic includes functional correctness from the user's perspective. Here, it is used for requirements where incorrect publication would directly mislead the user (e.g., wrong-language pages under a language-specific URL).

NFR-12 (Language-consistent URLs). Language-specific URLs shall accurately reflect the language of the delivered content. A language-specific route shall only be published if that route is available in the corresponding language. Direct requests to an unavailable language variant shall result in a redirect to the default language, and generated links shall not point to unavailable language variants.

Validation (Inspection): Verify that unavailable language variants are not generated in link outputs and verify the redirect for direct access to unavailable language paths.

3.5.5 Compatibility

This characteristic covers interoperability with other consumers of the output. For this thesis, the relevant consumer is a search crawler that should be able to extract primary content without client execution.

NFR-13 (Crawlable static output for eligible routes). Eligible routes shall output prerendered HTML that contains meaningful primary content and essential head metadata required for indexing.

Validation (Inspection): Inspect emitted HTML and verify that main content and head metadata are present without client execution.

NFR-14 (Compatibility with existing build toolchain and CI environment). The build and SSG run shall be executable using the existing gulp-based build orchestration and shall run within the established CI job container environment without requiring changes to the container image beyond project-managed dependencies and without requiring privileged execution.

Validation (Inspection): Inspect the build and CI configuration and verify that the build and SSG run are invoked as gulp tasks (or are callable via gulp), and the CI pipeline executes these tasks successfully in the existing job container image without additional system-level dependencies or elevated permissions.

3.5.6 Interaction Capability

Interaction capability broadly describes how well a system enables users to understand, learn, and control it through its interface so they can complete their tasks effectively across different contexts and user groups.

NFR-15 (Non-disruptive and accessible quality-state disclosure). Quality-state indicators and fallback disclosures should not block core navigation and should be dismissible while remaining accessible. The disclosure and its interactive controls shall be perceivable and operable via keyboard navigation and assistive technologies. In particular, all interactive disclosure elements shall expose an accessible name and role to screen readers, and icon-only controls shall provide an explicit accessible label.

Validation (Inspection): Inspect the disclosure element and verify that it is announced with an appropriate accessible name and that all disclosure controls are operable.

An overview of the non-functional requirements is provided in Table 3.2.

ID	**Short Title**	**ISO/IEC 25010 Characteristic**	**Validation**
NFR-01	Reduced SSG build time	Performance efficiency	Measurement
NFR-02	Parallel scalability	Performance efficiency	Measurement
NFR-03	Largest Contentful Paint on public pages	Performance efficiency	Measurement
NFR-04	Route-bounded translation payload	Performance efficiency	Measurement
NFR-05	Deterministic artifacts	Reliability	Inspection
NFR-06	Consistent behavior between SSG and CSR	Reliability	Inspection
NFR-07	Resilient failure handling	Reliability	Inspection
NFR-08	Route-to-namespace maintainability	Maintainability	Inspection + Test
NFR-09	Sustainable toolchain	Maintainability	Inspection
NFR-10	Runtime telemetry for translation loading	Maintainability	Inspection + Test
NFR-11	Developer documentation	Maintainability	Inspection
NFR-12	Language-consistent URLs	Functional suitability	Inspection
NFR-13	Crawlable static output for eligible routes	Compatibility	Inspection
NFR-14	Compatibility with existing build toolchain and CI environment	Compatibility	Inspection
NFR-15	Non-disruptive and accessible quality-state disclosure	Interaction capability	Inspection

Table 3.2: Overview of non-functional requirements

4 Architecture

This chapter describes the architecture and technical realization of the SSG build tooling, as well as the localization workflow and its integration into the existing QDAcity platform. The frontend is implemented as a React-based single-page application using React Router for navigation, while the production deployment serves the compiled frontend assets via *Google Cloud Platform* and *Spring Boot* as static resources. Build and automation tasks are orchestrated through the established tooling and executed within GitLab CI.

The proposed solution combines three concerns that must work consistently across the system: extracting and structuring translatable messages (based on FormatJS), synchronizing and managing translations through the translation workflow (including Weblate and optional machine-translation support), and packaging and publishing translation resources so they can be consumed reliably at runtime across both statically generated and client-side rendered routes. The following sections first establish the system context and boundary, then detail the architecture and implementation choices and their implications.

4.1 System Context

From a system context perspective, the solution comprises more than the React web application itself. The system-of-interest in this thesis includes the runtime delivery of localized content to end-users, and the toolchain and workflow that produces, validates, and publishes translation resources as part of the build and release process. Consequently, the relevant context spans the application runtime (browser and static asset delivery), the translation management workflow, and the automation pipeline that synchronizes translation artifacts between tooling and version control.

At runtime, the application executes in the user's browser and retrieves most static frontend assets, including translation bundles, via the Google Cloud Platform (GCP) delivery infrastructure. Through dedicated routing, these static resources are served before requests reach the application backend. All remain-

ing requests are handled by the existing *Spring Boot* backend. This separation reduces backend load and improves the delivery performance of static resources, while translation loading behavior, language resolution, and fallback handling remain the responsibility of the frontend application.

4.1.1 Actors

The primary actors interacting with the system are:

- **End-users**
  - Use the web application in various languages and expect the interface language to match the language encoded in the URL and the selected application language.
  - Navigate across statically generated and client-side rendered routes and expect consistent UI text, messages, and navigation elements across rendering modes.
  - Are affected by translation availability, fallback behavior, and user-facing disclosure mechanisms when a language is not fully translated.
- **Developers**
  - Implement and maintain the application codebase.
  - Introduce and evolve translatable messages and ensure new strings are extractable and correctly associated with the intended translation scope.
  - Maintain build and automation logic and ensure the resulting artifacts remain deterministic and deployable through the established pipeline.
- **Translators (Human)**
  - Provide localized strings and handle language-specific nuances.
  - Work primarily within the translation management system rather than directly editing repository files.
- **Reviewers / Approvers**
  - Review translations for correctness, consistency, and quality according to the project's workflow.
  - Control whether translations are considered eligible for release by approving changes within the translation management process.

- **CI/CD System**
  - Executes build-time and integration-time tasks such as extraction, validation, synchronization, interpolation, artifact packaging, and deployment steps.
  - Enforces quality gates and produces static artifacts consumed by the runtime.

4.1.2 External Systems and Services

The system interacts with several external systems and services:

- **Weblate (translation management system)**
  - Provides the collaboration surface for translation, review, and approval.
  - Exchanges translation artifacts with the repository through synchronization.
- **DeepL (machine translation service)**
  - Supports machine translation for pre-translation or suggestion workflows (primarily to accelerate throughput and reduce developer friction when new or changed messages are introduced).
  - Does not replace the review/approval process. Machine-generated content remains subject to human oversight.
- **GitLab (Repository and CI platform)**
  - Hosts the authoritative version-controlled source for application code and translation artifacts.
  - Runs CI pipelines that build the application, validate translation resources, and generate deployable artifacts and metadata.
- **Delivery infrastructure (Google Cloud routing and Spring Boot backend)**
  - Delivers the compiled frontend assets and translation bundles to the runtime clients.
  - Uses platform-level routing in Google Cloud Platform to intercept and serve a relevant subset of static assets.
  - Reduces backend load and improves response times for static resources delivery.

– Keeps translation loading, language selection, and fallback handling within the frontend application.

Figure 4.1 summarizes the involved actors and the external systems at the boundary of the system-of-interest.

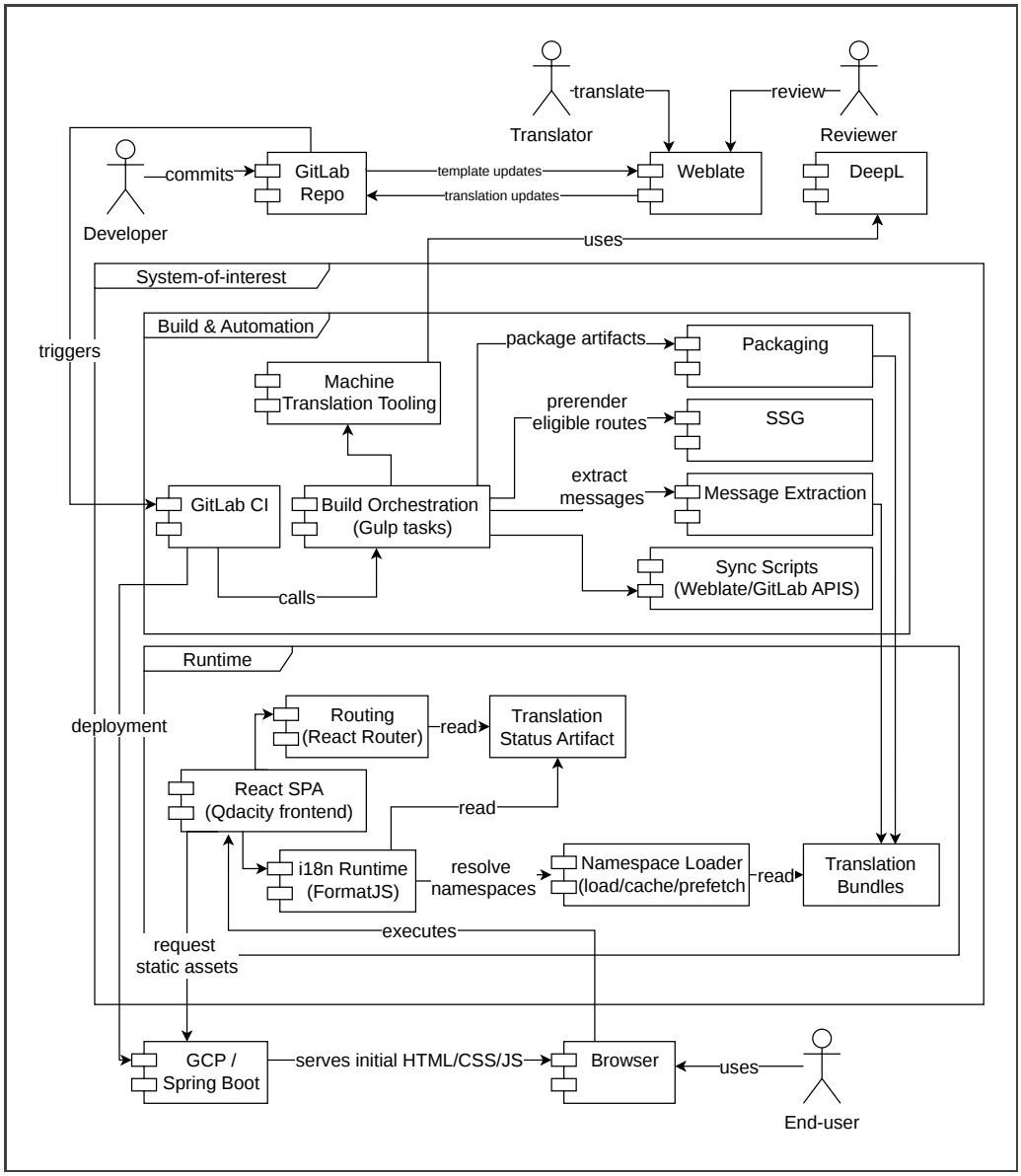

Figure 4.1: System Context Diagram (drawn with draw.io)

4.2 Baseline (As-Is) Architecture

This section summarizes the architecture of the system as it existed before the changes introduced in this thesis. The baseline already combines a React single-page application with selective static prerendering for suitable routes and multilingual UI support. In practice, however, two baseline design decisions become limiting as route count and language coverage increase: crawler-based prerendering, and monolithic, application-wide translation files per language. Both mechanisms address the immediate need for faster initial content display and localization, but they introduce scaling and evolution risks that directly motivate the requirements and target architecture in Chapters 1–3.

4.2.1 Crawler-Based Static Site Generation with React-Snap

For SSG, the baseline used React-Snap. React-Snap generates static HTML by executing the application in a headless browser and capturing a crawled snapshot of the rendered DOM for each configured route. The resulting HTML can be served directly and typically improves perceived performance compared to purely client-side rendering because the browser can present an initial document before JavaScript execution completes.

As the application grew, crawler-based generation showed two practical limitations. First, build duration tended to increase with the number and complexity of routes, because each route needed to be fully executed and rendered during prerendering. The crawler-based approach was also unfit for parallelization of snapshot generation. This increased CI resource consumption, slowed down feedback cycles, and made the build pipeline more expensive and more fragile as route volume grew.

Second, the snapshot result was sensitive to incidental runtime behavior. Redirects, asynchronous initialization, and navigation side-effects could influence what is captured. In practice, this could produce incomplete or unintended output for individual routes (e.g., missing content or snapshots that reflect transient intermediate states). Because the HTML was derived from a client-like runtime, such issues were often difficult to reproduce deterministically across build environments. Minor changes could therefore break a subset of snapshots and may only be detected late through manual inspection of output artifacts or production feedback.

These effects were particularly problematic in a hybrid rendering setting. The crawler-based HTML was generated under one concrete runtime configuration, while the client hydrated under potentially different conditions (e.g., locale initialization timing, deferred resource loading, or non-deterministic side effects during

startup). This increased the likelihood of inconsistencies between prerendered output and client-side behavior, such as missing content, mixed intermediate states, or hydration mismatch warnings.

Beyond these scaling and correctness concerns, the approach introduced an evolution risk. Snapshot output must remain compatible with the client-side hydration behavior of the React runtime. During attempted framework upgrades, pages generated with React-Snap could not be hydrated reliably under React 18 concurrent rendering, which effectively blocked upgrading React without replacing the prerendering approach. This kind of compatibility friction becomes more likely as routing structure, rendering semantics, or build tooling evolve. React-Snap itself is no longer actively maintained, judged by its last non-documentation update on GitHub dating back to 2022.[1]

Overall, the baseline prerendering approach worked adequately for smaller route sets, but it exhibited unfavorable scaling behavior in build-time cost, operational robustness, and framework evolvability as the application grew.

4.2.2 Monolithic Translation File Approach

Internationalization in the baseline was implemented with a **monolithic translation catalog per language**. Each language was represented by a single large merged message file that contained translations for the entire application. While this was straightforward to introduce initially, this structure became increasingly problematic as the application's frontend (and therefore the catalogs) grew.

From a runtime perspective, the monolithic structure coupled translation payload size to overall application size rather than to the user's navigation path. Even users who only interacted with a small subset of routes still downloaded and parsed a catalog that contained messages for unrelated features. For client-side rendered routes in particular, localization depended on the availability of the full catalog, which increased startup cost and delayed fully resolved UI text on slower devices or constrained networks. The result was unnecessary transfer and parsing work for messages that were never used in a given session.

From a workflow and maintenance perspective, a single shared file per language was difficult to review, prone to merge conflicts, and provided limited modular ownership boundaries. Translation changes across independent features were tightly coupled through one artifact, which reduced isolation between developers and increased coordination overhead. When several features evolved in parallel, developers and translators repeatedly touched the same files, which made change sets harder to review and caused translation updates and feature development to become operationally intertwined.

[1] https://github.com/stereobooster/react-snap (latest commit on master: 14 Nov 2022).

The monolithic granularity also complicated incremental language rollout. If a language is only partially translated, the system would either have had to ship an incomplete global catalog (risking mixed-language pages through fallback/default messages) or delay the language entirely, rather than enabling controlled, route-level publication. In practice, a new language therefore implied creating an application-wide catalog even if only a subset of routes was intended to be localized first. Without an explicit notion of *route availability per language*, the system would have tended to drift into ambiguous states: pages may have remained technically accessible under a language while lacking adequate translation coverage, or navigation may have linked users into under-translated areas without a clear eligibility rule.

In addition, application-wide catalogs caused strings for rarely used modules or restricted areas (e.g., administrative UI) to be shipped to all users. While this was not inherently a security vulnerability, it weakened the principle of shipping only what is necessary and may have exposed internal terminology or unreleased feature wording through shipped UI text.

Overall, the baseline translation approach was simple to introduce, but it did not scale well in runtime efficiency, team workflow, or controlled language rollout. As language coverage and frontend scope increased, these limitations motivated a more modular, route-aligned localization architecture with explicit publication rules per language.

4.3 Target (To-Be) Architecture

This section describes the target architecture introduced by this thesis. The target design addresses the baseline limitations by establishing a deliberate hybrid rendering model and restructuring internationalization into modular translation resources that can be loaded incrementally. In contrast to the baseline, where static generation and localization concerns are coupled to monolithic artifacts and crawler-based tooling, the target architecture makes route boundaries the primary unit for both rendering decisions and translation dependencies.

The architecture is structured along the four solution components defined in Chapters 1–3: controlled static generation for eligible routes, route-aligned translation namespacing, incremental translation with explicit states, and a governed translation workflow integrated via Weblate and CI. This section focuses on the structural aspects of the target system. The incremental rollout rules and user-facing implications are specified separately in section 4.4.

4.3.1 Architecture Goals

The target architecture is driven by the following goals:

- **Predictable hybrid rendering:** enable a controlled mix of statically generated (SSG) and client-side rendered (CSR) routes without relying on crawler-based SSG of the application.
- **Modular translation resources:** reduce translation payload size by loading only the translations required for the currently active route and shared application shell.
- **Incremental language adoption:** allow introducing a new language and expanding its coverage step-by-step instead of requiring a complete application-wide translation upfront.
- **Governed translation lifecycle:** integrate translation work (including machine translation and human review/approval) into a structured workflow with clear roles and quality states.
- **On-demand translation delivery:** load translation strings when the corresponding routes are accessed, rather than bundling all feature text into every initial page load.

4.3.2 Route-Centric System Structure

A central design decision of the target architecture is to treat routing as the architectural backbone. The route structure is defined in a dedicated configuration file (`routes.mjs`) alongside the application's route composition. Beyond defining paths and route hierarchies, this configuration also carries architectural metadata that is relevant for localization and delivery.

In particular, each route can declare a list of required translation namespaces via a `ns` property. The `ns` property defines which translation resource bundles must be available to render that route's UI correctly. The target architecture thereby establishes an explicit mapping between *routes* and *translation dependencies* without introducing a separate manifest artifact. Maintaining this mapping next to the routing configuration improves maintainability: when a route is added or changed, its translation dependencies are declared at the same boundary where the route is introduced.

The route configuration can also represent language-specific paths (e.g., different path segments per language). This supports localized URLs while keeping translation resource loading aligned with the active language. Routes that do not render user-facing content (e.g., redirects implemented via navigation primitives) can omit route-specific namespaces. The same applies to nested routes that fully reuse the parent route's namespace and therefore introduce no additional translation dependencies.

4.3.3 Hybrid Rendering Model

The target architecture supports a hybrid rendering approach in which selected routes are delivered as prerendered HTML (SSG), while the remainder of the application is rendered on the client (CSR). The key difference from the baseline is not that prerendering exists, but that it is applied in an explicitly configured way rather than as a snapshot of an implicitly crawled application state.

SSG is applied where it provides clear value, such as fast initial render for high-traffic public pages or routes with stable content structure. CSR remains the default for highly interactive application areas and for routes that depend on runtime context (e.g., authenticated state). This separation ensures that prerendering effort is invested where it is beneficial, while the system retains flexibility for dynamic parts of the product.

Another primary architectural constraint in a hybrid system is which pages are prerendered in the first place. In the target architecture, `routes.mjs` acts as the central source of truth for this decision: it defines which routes are considered public and indexable (and are therefore included in the sitemap). The same classification is reused to derive which routes are rendered via SSG versus CSR. Since indexable routes represent public information pages that benefit most from search engine visibility, they are preferentially prerendered. Importantly, SSG is performed only for the route's available languages, ensuring that no static crawlable resource is generated for a language variant that has not reached the required translation state. This keeps the set of statically generated pages consistent with the incremental translation rollout.

Hydration. Selected routes are delivered as static HTML but hydrated on the client to enable interactivity and client-side navigation (e.g., login, register, contact). To avoid hydration mismatches, the client bootstrap uses the same deterministic inputs as the SSG run (notably language and namespace resolution) and ensures required namespaces are available before hydration. All SSG routes ship with a minimal `publicPages.js` bundle that hydrates only the functionality required for public pages, avoiding loading the full SPA bundle. When accessed from an authenticated state, the application behaves as a fully hydrated SPA and all routes are available as interactive React views.

4.3.4 Translation Resource Architecture

The baseline approach of one monolithic translation file per language is replaced with a modular namespace structure. A namespace represents a cohesive subset of translations that can be loaded independently. In the target architecture, namespaces are organized primarily along route boundaries, complemented by shared namespaces that are used across multiple routes.

From a runtime perspective, namespaces are loaded for three distinct reasons:

- **Base namespaces:** a minimal set of namespaces required to render the application shell (e.g., global navigation, common actions, generic error messages).
- **Route-specific namespaces:** namespaces declared for the currently active route via the `ns` property in `routes.mjs`.
- **Prefetching for anticipated navigation:** additional namespaces that may be loaded in advance to reduce perceived latency for likely next navigations (discussed in subsection 4.3.8).

Runtime Namespace Loading

To make the route-bound namespace model operational at runtime, namespace activation follows navigation rather than a global initialization. When a user navigates to a route, the router resolves the active route and its required namespaces declared in `routes.mjs`. The namespace loader first checks the client-side cache. On a cache miss, it reuses an already in-flight request for the same namespace where possible. Otherwise, it fetches the missing bundle for the active language. The resolved messages are then merged into the active message set and passed to the FormatJS runtime for rendering. This keeps loading aligned with the route boundary while avoiding eager global loading and redundant requests. This message loading logic is illustrated in Figure 4.2.

This design improves performance by reducing the amount of translation data that must be transferred and parsed on startup. It also improves modularity: translations for isolated parts of the system can be placed into dedicated namespaces that are only loaded when those routes are actually visited. As a result, the system avoids distributing irrelevant or restricted feature text as part of a global catalog for every user session.

Operationally, namespace modularization also improves maintainability. Smaller translation units reduce merge conflicts, make reviews more focused, and allow clearer ownership boundaries. Furthermore, because namespaces are tied to navigation, the cost of localization scales with what a user accesses rather than with the total size of the application.

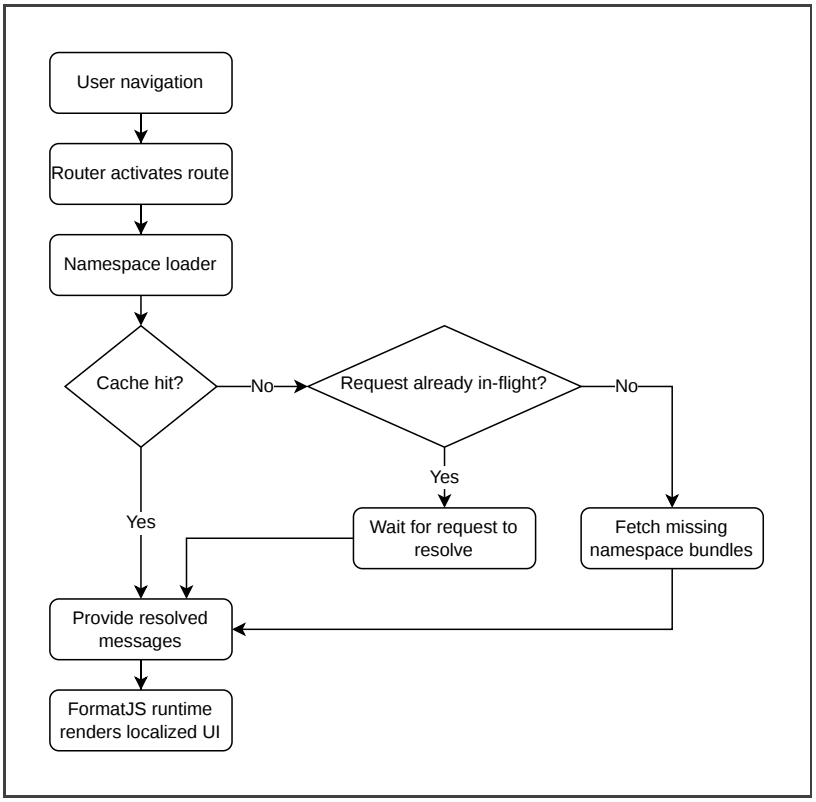

Figure 4.2: Architectural flow of namespace loading (drawn with draw.io)

Rationale for Route-Bound Namespaces

A central design decision of the target architecture is to align the translation namespaces primarily with routes. While alternative partitioning strategies exist (e.g., per component or folder-based grouping), route-bound namespaces were chosen because they match the system's runtime access pattern: translation resources are needed when a user *enters a route*, not when the application is built or when the codebase is loaded.

Route alignment provides three practical benefits. First, it makes translation dependencies *explicit and deterministic*: the route configuration declares required namespaces via the `ns` property, so the runtime can decide what must be loaded before rendering without heuristics or deep component graph analysis. Second, it supports the thesis goal of *incremental language rollout*: availability and quality can be evaluated at route granularity, preventing mixed languages per page and allowing for state management of certain routes. Third, route-bound splitting naturally limits upfront payload size, because translations for unrelated routes are deferred until the user actually navigates there.

Alternative: component-based namespaces. Managing translations at component level can maximize reuse, but it would lead to very fine-grained translation bundles. In practice, routes often consist of many shared and nested components, which would require loading a large number of small namespaces. This would increase request overhead and could lead to sequential loading during navigation, where namespaces are fetched one after another as components mount, accumulating avoidable latency. It also complicates correctness: ensuring that *all* component namespaces required for a route are present before render would require either static dependency graph tooling or runtime tracing, both of which increase architectural complexity and can become fragile as the UI evolves.

Alternative: folder-based namespaces. A folder-based structure reflects the code layout, but code organization does not necessarily match user journeys or runtime navigation boundaries. Refactorings (e.g., moving files between folders) may unintentionally reshuffle translation resources without semantic changes, increasing churn in translation artifacts. Furthermore, UI features are often cross-cutting (shared widgets, shared dialogs), which makes folder boundaries a weak proxy for translation dependencies and can result in either overly large bundles or frequent duplication.

Hybrid approach. The architecture therefore uses route-bound namespaces as the primary unit, complemented by a small shared *common* namespace for cross-route UI elements (e.g., navigation and generic actions). This keeps the number of namespaces manageable, preserves reuse where it is structurally stable, and maintains a direct mapping between navigation context and required translation resources.

4.3.5 Translation Lifecycle Integration and Quality Control

The target architecture treats translation as a governed lifecycle that is integrated into the development and deployment workflow. The lifecycle connects the codebase, translation management, and the CI pipeline in a structured sequence:

1. **Extraction:** translation keys and source strings are extracted from the codebase.
2. **Synchronization to Weblate:** extracted strings are synchronized to the translation management system (Weblate), where they become available to translators and reviewers.
3. **Machine translation support:** a machine translation service can provide suggestions or pre-translations to accelerate throughput, while maintaining human oversight for quality.

4. **Translation and review:** translators provide translated strings and reviewers validate them according to the project's quality process.
5. **Commit back to version control:** once changes are ready, Weblate commits updated translation artifacts and pushes them back to the repository.
6. **Merge request-based integration:** translation updates are integrated via merge requests, enabling automated checks (e.g., build and consistency validation) and human review before changes are merged into the main branch.
7. **Export and packaging:** after merging, translations are packaged into language- and namespace-specific bundles that are consumable by the runtime as part of the build output.

This lifecycle makes the chosen environment compliant with the requirements for incremental translation.

4.3.6 Build, Packaging, and Delivery

The target architecture produces deployable artifacts in two categories:

- **Frontend artifacts:** compiled application assets and where applicable, prerendered HTML outputs for SSG routes.
- **Translation artifacts:** per-language translation files split by namespace, enabling independent loading of route-specific resources.

These artifacts are built and assembled as part of the build pipeline, which coordinates extraction, validation, packaging, and deployment steps. The resulting build output is delivered through the existing backend delivery layer, which serves the compiled frontend assets and translation bundles to clients.

Importantly, this architecture keeps the delivery model stable while improving runtime behavior: the backend remains responsible for shipping static resources, while the frontend runtime controls which translation namespaces are fetched and when, based on route activation.

4.3.7 SEO Publication Artifacts and Re-indexing

In the target architecture, the publication model governs not only runtime navigation but also the set of URLs exposed to external discovery mechanisms. Consequently, search-engine-facing publication artifacts are treated as first-class outputs of the build and release process. Only route-language variants that satisfy the publication rules are exposed externally while unpublished language variants are excluded from discovery artifacts in the same way that they are excluded from language-specific navigation.

At build time, the system generates a sitemap from the same route definitions and language-availability information that also govern static generation and route publication. This ensures that the sitemap contains all and only published language-specific routes. For each published entry, the sitemap includes the corresponding `lastmod` value as well as the set of published localized alternates. As a result, crawler-facing output remains consistent with the route-gated localization model and does not expose language variants that are not yet intended to be publicly available.

The architecture deliberately separates deterministic artifact generation from external notification. Sitemap generation belongs to the build and packaging stage and therefore remains reproducible from a given set of inputs. By contrast, active re-indexing notification is treated as a post-release concern. After a successful production deployment, a dedicated CI task identifies the subset of published route-language variants whose emitted output changed compared to the previous release and submits only those URLs to IndexNow. This avoids coupling the build itself to external network-side effects while still enabling timely re-indexing of updated content.

This separation reinforces the consistency of the overall architecture: runtime publication rules, static output generation, sitemap contents, and post-release discovery notifications are all derived from the same notion of route-language eligibility. Therefore, a language variant that is not publishable is neither linked internally, nor emitted as a sitemap entry, nor submitted for re-indexing.

4.3.8 Hover-Based Namespace Prefetching

Route-scoped translation namespaces enable lazy loading, but they can introduce a short, user-visible delay during navigation when route-specific namespaces must be fetched before all UI text is available. To mitigate this effect, the target architecture includes hover-based namespace prefetching as part of the runtime behavior.

The implemented trigger follows a simple intent signal on desktop: when a user hovers over a navigational link, the runtime proactively resolves the translation namespaces required by the link target. Because namespace dependencies are declared in `routes.mjs`, the prefetch mechanism can determine the required bundles without additional heuristics about component usage or translation keys.

Prefetched namespaces are resolved through the same loading pipeline described in subsection 4.3.4. As a result, the existing cache and request deduplication mechanisms apply automatically, preventing redundant downloads and ensuring that prefetching does not introduce duplicate transfers.

Beyond hover intent, additional prefetch strategies (e.g., viewport-based signals,

route-adjacency heuristics, or predictive models) could be considered as optional enhancements. However, they are not required by the architecture presented in this thesis, since the implemented system already achieves the intended effect using the hover-based mechanism.

4.3.9 Summary

In summary, the target architecture introduces a route-centric structure in which routes become the primary unit for defining both rendering behavior and translation dependencies. The explicit declaration of namespaces alongside routes in `routes.mjs` replaces monolithic translation catalogs with modular, navigation-aligned bundles, enabling efficient runtime loading and reducing upfront translation payload by deferring route-specific resources until they are needed. Finally, hover-based namespace prefetching is integrated into the runtime to reduce perceived navigation latency, while more advanced prefetch strategies remain optional enhancements. In addition, the architecture extends the same publication logic to external discovery mechanisms by generating locale-aware sitemap entries, including `lastmod` and published alternates.

4.4 Incremental Translation

This thesis introduces an incremental localization model that avoids mixed-language pages without relying on runtime fallbacks. Instead of serving partially translated routes, the system uses explicit translation states per namespace and language, and uses these states to decide which language-specific routes are published at all. Consequently, the URL language and the rendered UI language are identical by design, since a route is only exposed for a language when its required translation resources are available.

4.4.1 Translation Lifecycle

Translation progress is tracked at the granularity of namespaces (aligned with route boundaries as described in subsection 4.3.4). For each namespace and language, the system derives one of three discrete states:

- `missing`: The namespace is incomplete in that language (i.e., not all messages are translated), or the namespace resource is absent.
- `translated`: All messages of the namespace are translated (100% translated).
- `approved`: The namespace is fully translated (100% translated) and meets a review threshold of at least 90% approved strings.

The separation between *translated* and *approved* allows the system to distinguish functional completeness from linguistic quality. In particular, pages can be made available once translation completeness is reached, while still enabling a differentiated UI disclosure for content that is likely auto-translated or not yet sufficiently reviewed.

4.4.2 Publication Rules at Route Level

The system does not publish a route for a language unless the language is eligible for that route. Eligibility is determined by the set of namespaces that the route requires:

- Each route declares a set of required namespaces.
- A language is eligible for that route if all required namespaces are in state *translated* or *approved*.

Routes that are not eligible are excluded from the language-specific route set and are therefore not reachable in that language. This design has two important consequences.

First, it eliminates the need for a user-facing runtime fallback strategy: if a language-specific route would require missing translation resources, the route simply does not exist in that language. Second, it guarantees language consistency: users cannot accidentally navigate to a page where only parts are translated, because partially translated routes are never exposed.

Defensive fallback for missing namespaces. A minimal fallback remains implemented at the namespace loader level: when a language-specific namespace resource is requested but not available, the English resource is loaded as a last resort. This is treated as a defensive measure for misconfiguration and unexpected drift during development, not as a mechanism that is expected to occur in normal production navigation.

4.4.3 Review options for unapproved translations

The architecture provides two complementary review options for namespaces that are already *translated* but not yet sufficiently *approved*. Together, these address efficient identification of review candidates and contextual verification of their correctness, thereby covering the requirements FR-LC-05 and FR-LC-06.

The first review option is provided directly through Weblate. Since approval status is tracked per string, Weblate offers a filtering option that only shows unapproved entries for a given language and namespace. This enables reviewers to focus specifically on strings that still require validation without being distracted

by already approved content. In this mode, review remains tightly integrated with the translation management workflow: corrections can be made directly in Weblate, and the approval state can be updated immediately once a string has been verified.

However, reviewing strings exclusively in Weblate has an important limitation: it presents translations primarily as isolated text entries. While source references are available, the actual application context in which a string appears is often difficult to reconstruct from the translation alone. As a result, issues such as misleading wording, layout problems, or context-dependent ambiguities may remain difficult to detect during purely text-based review.

To address this limitation, the target architecture introduces an additional translation debug mode in the application itself. In this mode, all strings that are not yet approved in the current language are visually marked in the rendered UI. This allows reviewers to navigate the application normally while immediately seeing which visible content still requires approval. Review can therefore take place in the actual usage context of the string, including its surrounding labels, interaction flow, page structure, and visual constraints. This is particularly useful for validating terminology consistency, assessing whether a translation fits the available space, and identifying cases in which the correct wording depends on nearby content.

The debug mode is designed as a review aid rather than as a separate source of truth. It does not replace the approval workflow in Weblate. Instead, it makes unapproved content discoverable in context and thereby guides reviewers to the relevant strings. The actual approval decision continues to be recorded in Weblate so that translation state remains centrally managed and can still be consumed deterministically by the publication logic described above.

Architecturally, the two review options complement each other. Weblate filtering provides a structured and efficient backlog view of all strings that still require review, whereas the translation debug mode provides contextual inspection inside the running application. The former is better suited for systematic translation management, while the latter is better suited for validating linguistic and UI-level appropriateness. Combined, they create a review workflow in which unapproved translations can first be located either centrally or contextually, then corrected and approved in Weblate, and finally promoted from *translated* to *approved* once sufficient review coverage has been reached.

4.4.4 User-Facing Disclosure for Auto-Translated Content

Because `translated` denotes completeness but not necessarily review quality, the UI provides explicit disclosure when a route is classified as auto-translated (i.e., translated but not sufficiently approved). This disclosure is implemented as a

sticky notice bar shown at the top of affected pages. It informs users that the page was automatically translated and may contain inaccuracies, and it offers a direct way to access the original (English) content.

Specifically, the notice bar:

- is shown only when the current route is marked as auto-translated for the current language,
- provides a *Show original* action that navigates to the English equivalent route,
- can be dismissed for the current session, or disabled persistently for the current language.

This mechanism makes the rollout practical: languages are automatically published as soon as the text is complete, while the UI remains transparent about quality differences until review coverage reaches the *approved* state.

4.4.5 Language-Change Hinting in Links

To reduce confusion during navigation, the link component marks links whose target language differs from the current one. In such cases, it displays a small indicator (e.g., a globe icon) and a tooltip clarifying that the destination opens in another language. The indicator is omitted for external links and can be disabled on a per-link basis when the language change is already obvious or when the annotation would add unnecessary visual clutter (e.g., icon-only links or language switchers).

Link resolution is publication-aware: if the target route is not available in the current language (i.e., it is not published due to missing required namespaces), the link automatically falls back to the default language variant instead of producing a non-existent language URL. Such links are treated as cross-language navigation and therefore receive the indicator and tooltip.

This hinting complements the publication rules: language changes remain possible and explicit, while users receive immediate feedback when a navigation action is expected to change the language context. In combination with publication gating, this ensures that links never lead to unavailable language routes and that any resulting language switch remains transparent to the user.

4.4.6 SEO Implications of Route-Gated Localization

The publication-based approach also aligns localization rollout with search engine indexing:

- Only published language-specific routes are included in the sitemap and are therefore indexable.
- Non-eligible routes (i.e., missing required namespaces) are neither linked internally as localized routes nor listed in the sitemap.

These publication constraints are operationalized not only in runtime navigation behavior, but also in build- and release-level discovery mechanisms, namely static site generation, deterministic sitemap generation and post-release re-indexing notifications, as described in Section 4.3.7.

As a result, crawlers see a coherent set of pages per language, and partially translated or unavailable pages are excluded from discovery. This reduces unnecessary crawling of unpublished language URLs and prevents indexing pages that would fall back to English.

4.4.7 Environment Parity

The incremental translation model does not rely on environment-specific switches. The same publication rules and disclosure behavior apply in development and production. "Production readiness" is thus expressed through translation state and route eligibility rather than a separate configuration: once namespaces are *translated*, routes can be published. As soon as namespaces are *approved*, the UI disclosure is removed.

4.5 Summary

This chapter introduced a route-centric architecture for optimizing localized content delivery. Starting from the baseline limitations of crawler-based prerendering and monolithic per-language message catalogs, the target design makes route boundaries the primary unit for both rendering decisions in a controlled hybrid SSG/CSR model and translation dependencies through explicitly declared namespaces. This restructuring enables predictable builds, selective loading of translation resources, and a clearer separation between public, indexable routes and dynamic application areas.

Building on this foundation, the chapter defined an incremental localization model that governs when language-specific routes are published. Translation progress is tracked per namespace and language using the states *missing*, *translated*, and *approved*. A route is published in a language only when all required namespaces are at least *translated*, ensuring that the language encoded in the URL always matches the rendered content and preventing mixed-language pages. The architecture also includes user-facing transparency mechanisms for incremental

rollout, including disclosures for unapproved translations and publication-aware navigation when localized route variants are unavailable.

Weblate and CI complete this architecture by synchronizing translation updates through reviewable merge requests, validating namespace contracts, and exporting status metadata that drives language availability and quality disclosure. The same publication logic is also extended to search-engine-facing artifacts: only published route-language variants are exposed through sitemap entries and localized alternates, keeping external discovery aligned with route-level translation eligibility. Finally, hover-based namespace prefetching reuses the same route-declared dependencies to reduce perceived navigation latency without introducing additional heuristics.

Chapter 5 turns these architectural choices into concrete implementation artifacts. It details the SSG implementation, the runtime namespace loader and caching model, the publication-aware routing and link behavior, and the Weblate-driven automation and tests that keep the system deterministic and scalable.

5 Design and Implementation

This chapter documents the concrete design decisions and implementation work produced in the context of this thesis. It translates the route-centric architecture introduced in Chapter 4 into implementation artifacts that can be integrated into a production React codebase without requiring invasive refactoring or a departure from a hybrid CSR/SSG delivery model.

The implementation follows the same solution decomposition used in the requirements and architecture: a custom static site generation that can execute the existing application in a Node.js environment and render eligible routes at build time, and a modular localization system based on route-bound namespaces, deterministic message loading, and explicit translation states for incremental rollout. In addition, the development lifecycle is extended through CI automation and translation workflow integration, so that localization changes remain auditable and deterministic.

The chapter is structured as follows. Section 5.1 describes the implemented SSG, including its deterministic environment emulation, module and asset compatibility layer, server-side rendering pipeline, and parallel orchestration model. Subsequent sections (Sections 5.2–5.4) implement the route-aligned localization runtime and the incremental translation behavior (publication gating, disclosures, link hinting), and integrate translation status and updates through Weblate and CI governance. These sections operationalize the requirements for language-consistent URLs, deterministic publication behavior, and sustainable tooling. Section 5.5 explains the design and implementation of the DeepL-based translation tooling, including placeholder-safe machine translation and category-specific instructions for consistent output. Finally, Section 5.6 summarizes the validation and tests that enforce these contracts and prepare the evaluation in Chapter 6.

5.1 Static Site Generation Tooling

5.1.1 Design Goals

The SSG was designed to satisfy the following goals:

- **Compatibility:** existing application code should run during SSG without invasive refactoring, even if it assumes a browser environment.
- **Determinism:** each route render must be isolated so that global side effects do not leak across pages.
- **Throughput:** rendering must scale to large route sets by parallelizing CPU-bound work.

Figure 5.1 provides a high-level overview of the SSG execution architecture.

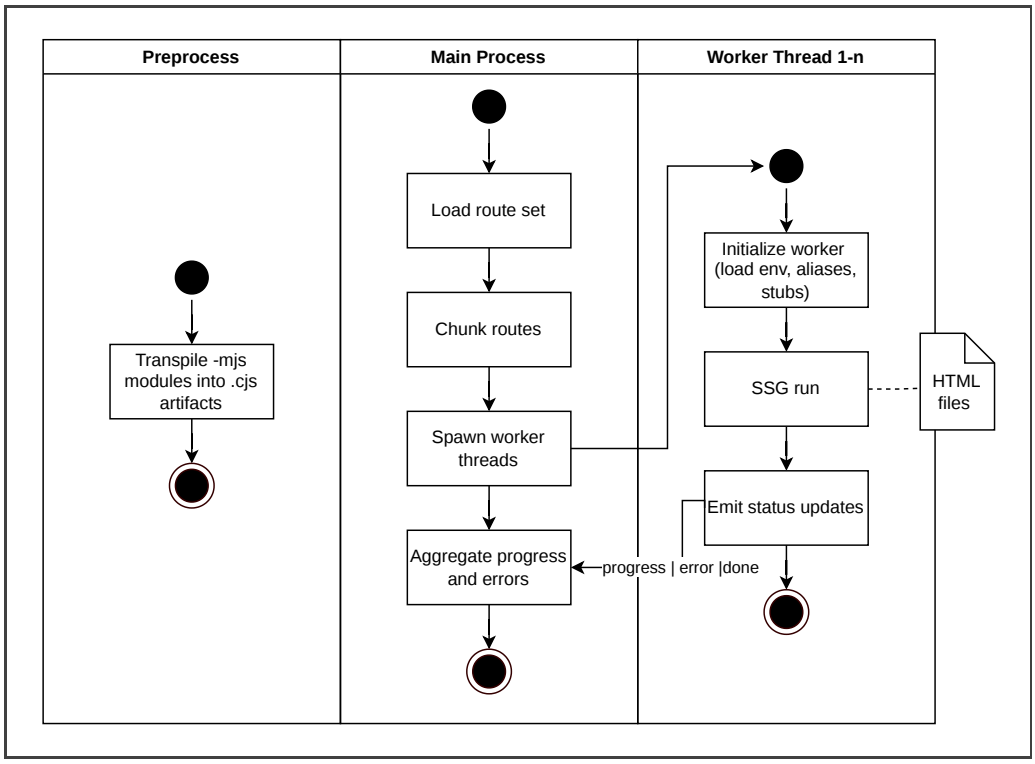

Figure 5.1: Execution architecture of the SSG (drawn with draw.io)

5.1.2 Environment Emulation for Node.js Rendering

The application is rendered through a Node.js script emulating a browser environment (FR-SSG-03). A `jsdom`-based DOM is instantiated and assigned to `global.window` and `global.document`. A mock `navigator` is provided (including language defaults), and browser-only APIs that commonly appear in production code are stubbed (e.g., `IntersectionObserver`). To support code paths that depend on runtime parameters, global configurations like frontendParametrization should also be mocked.

To make language selection stable during SSG, `localStorage` is implemented as

an in-memory store. This ensures that localization utilities and other utilities relying on persisted state can run without throwing, while still yielding deterministic results.

Finally, the environment exposes an explicit `window.isSSG` flag. This flag is used by the message loader to switch from network-based message loading to synchronous, file-based message loading during static prerendering (Section 5.2.3).

A shortened version of the environment setup code is shown in Figure 5.2.

```
module.exports = function setupEnv() {
    const { JSDOM } = require("jsdom");
    const dom = new JSDOM(
        `<body><div id="app"></div></body>`,
        { url: "http://localhost" }
    );

    global.window = dom.window;
    global.document = dom.window.document;
    global.window.isSSG = true;

    Object.defineProperty(global.window, "navigator", {
        value: {
            userAgent: "SSG",
            language: "en-US",
            languages: ["en-US", "en"],
            // more navigator properties
        },
        configurable: true,
        writable: true,
        enumerable: true,
    });

    const mockStorage = {};
    global.localStorage = {
        getItem: (key) => mockStorage[key] || null,
        setItem: (key, value) => {
            mockStorage[key] = String(value);
            },
        removeItem: (key) => { delete mockStorage[key]; },
        clear: () => {
            for (let key in mockStorage)
                delete mockStorage[key];
        },
    };
    //additional needed mocks and stubs
};
```

Figure 5.2: SSG Environment Setup Code

5.1.3 Module and Asset Compatibility

Static rendering must handle two categories of compatibility issues: module format mismatches and non-JavaScript assets (FR-SSG-04).

MJS-to-CJS compilation. Parts of the frontend codebase are authored as ECMAScript modules (`.mjs`), whereas the SSG runtime executed in a Node.js environment expects CommonJS `require()` semantics for dynamic imports. To bridge this mismatch, a dedicated preprocessing script traverses the repository and transpiles each `.mjs` module into a sibling `.cjs` file using a modules-to-CommonJS transform. During SSG execution, `require()` calls that resolve to `.mjs` paths are redirected to the corresponding `.cjs` artifacts. The generated `.cjs` files are excluded from version control by adding them to `.gitignore`, ensuring they remain a build artifact rather than repository content.

Alias parity with the bundler. To keep resolution behavior equivalent to the browser bundle, the same path aliases used in the application build are registered in the SSG runtime. Additionally, critical runtime configuration modules (such as routes and supported languages) are explicitly mapped to their `.cjs` counterparts.

Asset stubbing and SVG inlining. In order to prevent render-time failures, stylesheets (`.css`, `.scss`) are stubbed to empty exports. Binary assets (e.g., images, fonts, PDFs) are mapped to stable `/static/...` paths. SVGs are treated as a special case: instead of being mapped to a static path, they are inlined as `...` Uniform Resource Identifiers (URIs) to preserve visual output in the prerendered HTML.

Selective library stubbing. Certain modules that are browser-specific or irrelevant during static rendering (e.g., animation libraries, swagger-client) are replaced with no-op stubs. This ensures that static rendering follows the same component tree without requiring route-specific code forks.

5.1.4 Static Site Generation

The static rendering pipeline implements the core behavior specified in FR-SSG-01 and FR-SSG-02, which require static HTML creation for eligible routes and for all language variants in which the route is available.

For each route, a SSG run executes the following steps:

1. **Route rendering:** the application is rendered using the `renderToString` function from `react-dom/server` with a server-side router (`StaticRouter`) set to the target route location.
2. **Head collection:** `react-helmet-async` collects dynamic head metadata and attributes (HTML and body attributes).

3. **CSS collection:** the CSS code of `styled-components` is collected using `ServerStyleSheet` and injected into the final HTML.
4. **FontAwesome reduction:** the FontAwesome CSS bundle is loaded and then reduced using `PurgeCSS`, based on the actually rendered HTML content. A small safelist prevents removal of known dynamic classes.
5. **HTML assembly:** the base HTML template is updated with collected head tags, injected styles, and the rendered markup inserted into the mount element.
6. **File mapping:** routes are mapped to `index.html` outputs using a clean URL strategy: `/` produces `index.html` while any other routes produce `<route>/index.html`.
7. **HTML minification:** the final HTML is minified using `html-minifier-terser`[1] to reduce file size.

Isolation and cleanup. To ensure isolation between route renders, timers created by application code are tracked by wrapping `setTimeout`, `setInterval`, and `setImmediate`. After each route render, all tracked handles are cleared. This prevents asynchronous side effects from accumulating and preserves determinism across a large render set (FR-SSG-05).

A pseudo-code outline of a SSG run is shown in Figure 5.3.

[1] https://github.com/terser/html-minifier-terser

```
async function renderRouteToHtml(routes, config) {
    setupEnvironment();
    registerAssetStubs(); // stub static assets
    registerAliases(); // register path aliases
    registerBabel(); // enable JSX/TSX transpilation in Node.js

    installTrackedTimers(); // for cleanup after each render

    const App = require(config.appEntryPoint);

    for (const route of routes) {
        const context = {};
        const helmetContext = {};
        const sheet = new ServerStyleSheet();
        const element = React.createElement(
            StaticRouter,
            { location: route, context },
            React.createElement(
                HelmetProvider,
                { context: helmetContext },
                React.createElement(App)
            )
        );
        const html = ReactDOMServer.renderToString(
            sheet.collectStyles(element)
        );
        purgeCss(); // reduce FA CSS based on html content
        const {helmet} = helmetContext;
        const headInject = assembleHead(helmet, sheet);
        const finalHtml = injectIntoTemplate(html, headInject);
        writeOutputFile(route, finalHtml);
        clearTrackedTimers(); // cleanup timers to avoid leakage
    }
}
```

Figure 5.3: Static Site Generation Logic

5.1.5 Parallel Route Rendering

Rendering many routes is CPU-bound, as it involves JavaScript execution and server-side React rendering. The implemented SSG therefore parallelizes route rendering using Node.js `worker_threads` (NFR-02).

- The full route list is split into evenly sized chunks.
- A configurable number of workers is spawned, each receiving a chunk of routes.
- Each worker runs the same SSG code and writes its output files into a shared distribution directory.
- The main process aggregates progress reports and errors, and fails the build if any worker fails.

This design improves throughput on multi-core systems while keeping each worker

independent and reducing the risk of global state interference across route renders. The logic of the SSG orchestration is shown in Figure 5.4.

```
1. Prep
    - Ensure runtime/module setup is correct

2. Plan
    - Split all routes into N chunks (N = number of workers)
    - Initialize counters: processed, finishedWorkers, hadError

3. Execute
    - For each chunk:
    - Start a worker with that chunk + shared config
    - On "progress": increment processed, print progress
    - On "error/crash": mark hadError, log details
    - On "done/exit": mark that worker finished (only once)

4. Finalize
    - When all workers are finished:
    - Log total duration
    - If any worker failed -> reject/fail
    - Else -> resolve/succeed
```

Figure 5.4: Parallel SSG Orchestration Logic

5.1.6 Sitemap generation and `lastmod` derived from static HTML hashes

As discussed in the SEO implications of route-gated localization (Section 4.4.6), the sitemap must reflect the same publication rules as runtime navigation: only language-specific routes that are actually published should be discoverable and therefore indexable. This requirement is formalized in FR-INC-08, which specifies that production releases should generate a sitemap that includes only published language-specific routes and should submit it to configurable endpoints.

Sitemap as a deterministic build artifact. The sitemap is generated as part of the build pipeline after static HTML creation. Rather than enumerating URLs from a separate configuration, the generator derives its URL set from the route tree (`routes.mjs`) and applies the same constraints used for static generation: routes with dynamic path segments are excluded, and routes marked as `noindex` are omitted. This keeps the sitemap consistent with the set of public, crawlable pages and prevents accidental indexing of application-internal or parameterized routes.

To align with publication gating, the generator emits one `<url>` entry per English public page and attaches `hreflang` alternates for the languages in which the same route is considered fully translated (i.e., published for that language). This

directly operationalizes the architectural rule that non-eligible language variants must neither be linked nor listed in the sitemap.

Concretely, alternate links are filtered such that only languages that satisfy the route-level availability predicate contribute an `xhtml:link rel="alternate"` entry. As a result, crawlers observe a coherent set of language variants and do not discover URLs that would otherwise fall back to the default language.

`lastmod` via static HTML hashing. A naive strategy for setting the `lastmod` (e.g., setting all URLs to the current build date) can mislead crawlers and reduce the value of the signal. For this reason, `lastmod` is derived from content change detection on the generated static HTML files.

For each public English route, the build locates the corresponding generated HTML file and computes a SHA-256 checksum over the document body. To avoid noise from head-level changes that are not content-relevant (e.g., build-time injected tags, reordered metadata), the checksum is computed only for the substring following the closing `</head>` tag. The resulting hash represents the rendered page content and is stable across builds unless the visible output changes.

Hashes and `lastmod` dates are persisted in a JavaScript Object Notation (JSON) report keyed by route path. On each build, the report is updated as follows:

- If a route already exists in the report and its hash has changed, the entry is updated and `lastmod` is set to the current build date.
- If a route exists and the hash is unchanged, `lastmod` is preserved (and may remain absent), while the stored hash is refreshed.
- If a route is new (no previous report entry), `lastmod` is only set if the hash differs from a baseline checksum. Otherwise it is omitted to avoid claiming a fresh modification for pre-existing content.

Baseline-aware initialization. To prevent the first run from marking the entire sitemap as newly modified, the generator supports baseline-aware initialization by comparing computed hashes to a baseline report downloaded from the existing build quality infrastructure (e.g., the Lighthouse CI `master.json` report). If the baseline already contains a checksum for a given route and it matches the current HTML body hash, the route is initialized without a `lastmod` value. This yields conservative behavior: only pages that truly differ from the established baseline receive a `lastmod` timestamp on first introduction of the mechanism.

Persistence and production behavior. The `lastmod` report is written into the build output directory and, on release, uploaded to a persistent bucket location so that subsequent releases can perform a stable comparison against the

previous state. When the report cannot be retrieved, the build continues without failing and emits a sitemap without `lastmod` information for the affected URLs. This preserves local developer ergonomics while keeping production releases deterministic and comparable.

Relation to sitemap submission (IndexNow). The implementation in this section intentionally separates deterministic sitemap generation as a build artifact from sitemap submission as a release automation concern. Accordingly, *IndexNow* verification (key file) is produced during packaging, while actual submission is performed in the CI/CD stage responsible for post-release deployment and external notifications (described in Section 5.4). The key file generation logic is shown in Figure 5.5.

```
async function createIndexNowKeyfile() {
    const indexNowKey = process.env.INDEX_NOW_KEY;
    if (!indexNowKey) {
        console.debug('No INDEX_NOW_KEY provided. Skipping creation of
    indexnow keyfile.');
        return;
    }
    const keyFilePath = `./${DIST_DIR}/${indexNowKey}.txt`;
    fs.writeFileSync(keyFilePath, indexNowKey, 'utf8');
}
```

Figure 5.5: IndexNow Key File Generation Logic

5.2 Modular Localization Architecture

The localization runtime implements the route-aligned, namespaced message architecture introduced in Chapter 4. In contrast to monolithic per-language catalogs, translation resources are split into independently loadable namespaces and bound to navigation boundaries through route metadata in `routes.mjs`. This design allows the runtime to load only the baseline application shell translations and the namespaces required by the currently active route, while retaining deterministic behavior during static site generation (Section 5.1) via filesystem-based loading.

5.2.1 Namespace Splitting and Route Binding

Namespaces are declared directly in the route configuration. Each route entry in `routes.mjs` can specify a `ns` property containing a list of namespace slugs (e.g., `['about']`, `['landing-page']`). This keeps translation dependencies co-located with the navigation boundary where they become relevant, avoiding a

separate manifest and reducing drift between routing and localization configuration. Routes that do not introduce user-facing content (e.g., redirects) can omit route-specific namespaces and therefore inherit only the baseline namespace behavior.

An excerpt of the route configuration with namespace declarations is shown in Figure 5.6.

```
const routes = [
    {
        name: 'public-pages',
        paths: '',
        children: [
            {
                name: 'googleCampaign',
                paths: 'ga/:campaignId',
            },
            {
                name: 'about',
                paths: { en: 'about', de: 'ueber-uns' },
                ns: ['about'],
            },
            {
                name: 'core',
                paths: '*',
                ns: ['core'],
                children: [
                    {
                        name: 'projects-dashboard',
                        paths: 'projects-dashboard',
                    },
                ],
            },
            // ...
        ],
    },
];
```

Figure 5.6: Example route configuration with namespace declarations

The runtime resolves the required namespaces for a concrete URL path by matching it against the React Router route definition and collecting all `ns` entries along the match chain. The helper shown in Figure 5.7 performs this resolution and appends the shared baseline namespace `common`. In addition, another helper function provides the same aggregation for route names. This yields a deterministic, route-scoped namespace set that is reused across message loading, validation, and publication-aware routing decisions.

```
const getTranslationNamespacesForRoutePath = (routePath) => {
    const matchedRoutes = _matchRoutes(reactRouterRoutes, routePath);
    if (!matchedRoutes) return [];
    const namespaces = matchedRoutes.flatMap((r) => r.route.ns || []);
    return Array.from(new Set([...namespaces, 'common']));
};
```

Figure 5.7: Implementation of the route-to-namespace resolution logic

A naming convention ensures that message identifiers remain statically attributable to namespaces. Route metadata and translation bundle file names use slug-form namespace identifiers (e.g., `landing-page`), while message identifiers use a camelCase prefix derived from that slug (e.g., `landingPage.title`). Build-time checks validate that translation files adhere to this scheme by ensuring that every identifier inside `<namespace>.<lang>.txt` starts with the expected camelCase prefix, and that files follow a strict syntax (including a terminating newline and the required format).

To prevent accidental cross-route coupling, an additional build-time validation derives the "allowed" namespace prefixes for each route from `routes.mjs` and verifies that all message identifiers used in the transitive module closure of that route's entry components are within that allowed set. This check is implemented as a static crawl: starting from route entry components discovered in the application router, the validator resolves local imports, extracts message IDs, and reports mismatches where an identifier prefix is not declared by the route. The resulting validation enforces that namespace boundaries remain aligned with routes. This validation is critical for maintaining the integrity of the modular architecture and ensuring that the benefits of route-aligned namespaces are preserved over time.

5.2.2 Namespace Bundle Format and Lifecycle Hooks

Translation artifacts are packaged as JSON files per namespace and language. During the build, FormatJS extraction produces an English template (`en.json`) that contains all discovered message IDs and default messages. This global English file is then split into namespace-specific English bundles by grouping message IDs by their prefix and writing the grouped messages into files following the convention `<namespaceSlug>.en.json`. For production builds, an additional `<namespaceSlug>.en.txt` template is emitted for translation workflow integration (Section 5.4), but the runtime-facing format remains the JSON bundle per namespace and language.

Figure 5.8 shows the implementation of the Gulp task that performs the splitting of the English message extraction into namespace-specific bundles.

```
function splitEnglishMessageExtraction(cb) {
    const messagesDir = `./${BUILD_DIR}/messages`;
    const enSourcePath = `${messagesDir}/en.json`;
    const translationsRoot = './translations/';

    const enMessages = JSON.parse(fs.readFileSync(enSourcePath, 'utf8'));
    const groupedByNamespace = {};

    for (const [id, msg] of Object.entries(enMessages)) {
        const dotIdx = id.indexOf('.');
        const namespaceCamel = id.slice(0, dotIdx);
        const namespaceSlug = camelCaseToSlug(namespaceCamel);

        if (!groupedByNamespace[namespaceSlug]) {
            groupedByNamespace[namespaceSlug] = {};
        }
        groupedByNamespace[namespaceSlug][id] = {
            defaultMessage: msg.defaultMessage
        };
    }

    for (const [nsSlug, nsMessages] of Object.entries(groupedByNamespace)) {
        const targetPath = `${messagesDir}/${nsSlug}.en.json`;
        fs.mkdirSync(path.dirname(targetPath), { recursive: true });
        fs.writeFileSync(
            targetPath, JSON.stringify(nsMessages, null, 2), 'utf8');

        if (process.env.NODE_ENV === 'production') {
            const englishFilePath = `${translationsRoot}/${nsSlug}.en.txt`;
            const englishFileContent =
                Object.entries(nsMessages)
                    .map(([id, msg]) => `${id}=${msg.defaultMessage}`)
                    .join('\n') + '\n';
            fs.writeFileSync(englishFilePath, englishFileContent, 'utf8');
        }
    }
    cb();
}
```

Figure 5.8: Gulp Task for namespace splitting

Non-English translations are stored in text files `<namespaceSlug>.<lang>.txt`, where each line encodes an identifier/value pair (`id=message`). At build time, a dedicated parser converts these files into JSON bundles that map message IDs to message objects. The runtime consumes these compiled bundles in both CSR and SSG modes. The difference between rendering strategies lies in how the bundles are accessed (network fetch vs. filesystem require).

Namespace activation is performed through explicit lifecycle hooks in the React runtime. A `TranslationNamespacesProvider` maintains a session-level set of "active" namespaces, seeded with a configurable baseline. On route changes, the provider derives the namespace list for the current `location.pathname` via `getTranslationNamespacesForRoute` and merges it into the active set. Because the active set is monotonic (namespaces are added but not removed), revisiting

routes does not require re-fetching bundles, and prefetching can safely populate the same cache without additional coordination.

In addition to route-derived activation, a `TranslationNamespaceScope` component provides a local hook for explicitly adding namespaces in cases where the loading is not dependent on the route, e.g. prefetching. The scope component deduplicates its declared namespaces and adds them via the provider API. This mechanism retains route alignment as the default while allowing targeted exceptions without weakening the contract.

Figure 5.9 shows the implementation of the `TranslationNamespacesProvider` and `TranslationNamespaceScope` components.

```
const TranslationNamespacesContext = createContext(null);

function TranslationNamespacesProvider({ baseline = ['common'], children }) {
    const [active, setActive] = useState(() => new Set(baseline));
    const location = useLocation();

    const add = useCallback((namespaces = []) => {
        if (!namespaces.length) return;
        setActive((prev) => {
            const next = new Set(prev);
            let changed = false;
            namespaces.forEach((ns) => {
                if (ns && !next.has(ns)) {
                    next.add(ns);
                    changed = true;
                }
            });
            return changed ? next : prev;
        });
    }, []);

    useEffect(() => {
        const nsList = getTranslationNamespacesForRoute(location.pathname) ||
    [];
        if (nsList.length) {
            add(nsList);
        }
    }, [location.pathname, add]);

    const value = useMemo(() => ({ active, add }), [active, add]);
    return <TranslationNamespacesContext.Provider value={value}>{children}</
    TranslationNamespacesContext.Provider>;
}

function useTranslationNamespaces() {
    const ctx = useContext(TranslationNamespacesContext);
    if (!ctx) throw new Error('useNamespaces must be used within <
    TranslationNamespacesProvider>');
    return ctx;
}

function TranslationNamespaceScope({ namespaces = [], children }) {
    const { add } = useTranslationNamespaces();
    const key = React.useMemo(() => [...new Set(namespaces)].sort().join('|'),
     [namespaces]);
    React.useEffect(() => { if (namespaces.length) add(namespaces) }, [key,
    add]);
    return <>{children}</>;
}

TranslationNamespacesProvider.propTypes = {
    baseline: PropTypes.arrayOf(PropTypes.string),
    children: PropTypes.element,
};

TranslationNamespaceScope.propTypes = {
    namespaces: PropTypes.arrayOf(PropTypes.string),
    children: PropTypes.element,
};
```

Figure 5.9: Implementation of TranslationNamespacesProvider and TranslationNamespaceScope

5.2.3 Message loading (CSR and SSG)

The localization runtime is centered around a custom `IntlProvider` that integrates React Intl with the route-scoped namespace loader. The provider exposes an `IntlContext` interface that extends the standard `react-intl` shape with locale state, supported language/region sets, and routing helpers (e.g., `getRoutePath`, `getPathPattern`, `generatePath`). The provider also implements the message loading logic, which differs between CSR and SSG modes to satisfy the requirement for deterministic, network-independent loading during static generation (FR-I18N-04).

In CSR mode, message loading is performed through an asynchronous loader `loadMessages(language, namespaces)`. The `IntlProvider` maintains the current message map in component state and initializes the loaded messages with `window.defaultMessages` if present. When the active language changes, or the active namespace set changes due to navigation, the provider calls `loadMessages` with the effective namespace set derived from `useTranslationNamespaces()`. The resulting message map is then provided to react-intl's `IntlProvider` component.

During SSG, message loading must not depend on network access (FR-I18N-04). This is implemented through an explicit environment flag introduced in the SSG setup (Figure 5.2): `window.isSSG`. When the flag is set, the `IntlProvider` bypasses the network loader and instead synchronously reads compiled JSON bundles from the local filesystem under `build/messages`. The implementation resolves all files in that directory that match the requested language suffix and merges them into a single message map.

This loading path is intentionally synchronous to preserve determinism and to avoid introducing asynchronous rendering dependencies into the SSG. The provider also suppresses the CSR re-fetching effect during SSG, since the message map has already been seeded from filesystem artifacts. This ensures that route rendering can proceed without awaiting network requests and without nondeterministic timing effects.

The implementation of the IntlProvider is provided in Appendix A.

5.2.4 Caching, deduplication, and preload

The message loader implements caching and request deduplication at the granularity of (language, namespace), as required by FR-I18N-03.

`loadMessages` maintains:

- a per-language message map that stores the merged message map for that language.

- a per-language set of already loaded namespaces.
- an in-flight request map keyed by `"<lang>/<ns>"` to deduplicate concurrent loads.

When `loadMessages` is invoked, it first filters the requested namespace list to those not yet marked as loaded for the language. In order to deduplicate in-flight requests, `fetchNamespace` either returns an existing in-flight promise or initiates a new fetch to `/messages/<ns>.<lang>.json`. Successfully loaded bundles are merged into the language cache, and the namespace is recorded as loaded, ensuring that subsequent requests avoid redundant downloads.

A defensive fallback is implemented at the namespace level: if a namespace bundle for a non-default language is missing or fails to load, the loader retries the same namespace in the default language (`en`). If the default-language bundle is missing as well, the loader returns an empty object for that namespace. This behavior is explicitly treated as a safeguard against misconfiguration and drift, rather than as a normal production strategy, and is accompanied by warning logs.

The implementation of the loader is provided in Figure 5.10. This code also features a debug mode that is explained in Section 5.3.4.

There are two mechanisms implemented for ensuring that needed messages are available on initial render. First, the baseline namespace `common` is always active due to the provider's baseline configuration, ensuring that the application shell can render with consistent global UI text. Second, the `IntlProvider` seeds its initial state from `window.defaultMessages` (CSR) or from precompiled artifacts (SSG), enabling the initial render to start with a populated message map. The initial state (`window.defaultMessages`) for CSR is loaded by fetching it in the `index.js` entry point, before the React tree is rendered, ensuring that all messages are available for the first render.

A key trade-off of the chosen caching model is that the active namespace set and the per-language cache are session-global and monotonic: once a namespace is loaded, it remains available for the remainder of the session. This reduces repeat network traffic and makes prefetching straightforward, but it also implies that long sessions with broad navigation accumulate larger in-memory catalogs. No eviction policy is implemented in the provided runtime code, since the current application scope and usage patterns do not indicate a significant risk of unbounded growth, but this is an area for future enhancement if memory constraints become a concern.

```
const messageCache = new Map();
const loadedTranslationNameSpaces = new Map();
const inFlightRequests = new Map();

export const loadMessages = async (language = 'en', namespaces = ['common']) => {
    const basePath = '/messages';
    const fallbackLang = 'en';
    !messageCache.has(language) && messageCache.set(language, {});
    !loadedTranslationNameSpaces.has(language) && loadedTranslationNameSpaces.set(language, new Set());
    const debugParam = new URLSearchParams(window.location.search).get('debugTranslation');
    if (debugParam === 'true') sessionStorage.setItem('debugTranslation', debugParam);
    const debugEnabled =
        debugParam === 'true' || (debugParam === null && sessionStorage.getItem('debugTranslation') === 'true');
    const langCache = messageCache.get(language);
    const loadedForLang = loadedTranslationNameSpaces.get(language);

    const fetchNamespace = async (lang, ns) => {
        const key = `${lang}/${ns}`;
        if (inFlightRequests.has(key)) return inFlightRequests.get(key);
        const promise = (async () => {
            const url = `${basePath}/${ns}.${lang}${debugEnabled ? '.debug' : ''}.json`;
            try {
                const response = await fetch(url);
                if (!response.ok) {
                    throw new Error(`Failed to load ${url}: ${response.status} ${response.statusText}`);
                }
                return await response.json();
            } catch (err) {
                console.warn(`Missing namespace ${ns} for ${lang}, trying fallback (${fallbackLang})`);
                if (lang !== fallbackLang) {
                    return fetchNamespace(fallbackLang, ns);
                }
                return {};
            } finally {
                inFlightRequests.delete(key);
            }
        })();
        inFlightRequests.set(key, promise);
        return promise;
    };

    const nsToLoad = namespaces.filter((ns) => !loadedForLang.has(ns));
    await Promise.all(
        nsToLoad.map(async (ns) => {
            const msgs = await fetchNamespace(language, ns);
            Object.assign(langCache, msgs);
            loadedForLang.add(ns);
        })
    );
    return langCache;
};
```

Figure 5.10: Implementation of the message loader with caching and deduplication

5.2.5 Predictive prefetching

Route-scoped namespaces enable lazy loading, but they can introduce user-visible delay when navigation activates a route whose namespaces have not yet been fetched. The architecture therefore includes a prefetch strategy that reuses the same route-declared namespace dependencies, but shifts the load earlier in time based on user intent signals.

The primary mechanism described in the architecture is hover-based namespace prefetching on navigational links. When a user hovers a link, the runtime resolves the target URL to its route, obtains the corresponding namespace list via `routes.mjs` (`getTranslationNamespacesForRoute`), and triggers an opportunistic `loadMessages(currentLanguage, namespaces)` call. Because the loader is cache-aware and deduplicates concurrent requests, prefetching integrates without additional coordination: already loaded namespaces are ignored, and overlapping prefetch and navigation requests collapse into the same in-flight promise.

The prefetching is designed to be optional and non-blocking: navigation remains correct without it, while prefetching reduces perceived latency by increasing the probability that route namespaces are already resident at click time.

The implementation of hover-based prefetching is provided in Figure 5.11.

```
1     useEffect(() => {
2         function handleMouseOver(event) {
3             const link = event.target.closest('a[href]');
4             if (!link) return;
5             const href = link.getAttribute('href');
6             const namespaces = getTranslationNamespacesForRoute(href) || [];
7             if (namespaces.length) {
8                 add(namespaces);
9             }
10        }
11        window.addEventListener('mouseover', handleMouseOver, true);
12        return () => {
13            window.removeEventListener('mouseover', handleMouseOver, true);
14        };
15    }, []);
```

Figure 5.11: Implementation of Hover-based prefetching.

This completes the runtime foundations for namespaced, route-aligned localization. Section 5.3 builds on this foundation by introducing incremental translation behavior and publication-aware routing, which consumes the same route-to-namespace mapping to decide when language-specific routes are available and how navigation behaves under incomplete translation coverage.

5.3 Incremental Translation Implementation

Building on the namespaced localization runtime established in Section 5.2, this section describes how incremental translation is operationalized in the implementation. The objective is to enable incremental rollout of languages without introducing mixed-language pages or relying on user-facing runtime fallbacks. As specified in Section 4.4, the system therefore models translation progress as discrete states at namespace granularity, derives route-level availability from the route-to-namespace mapping, and couples navigation, disclosure, and SEO metadata to these availability decisions. In contrast to approaches that generate language-specific route trees by removing unavailable routes, the provided implementation enforces publication gating primarily at path generation and direct-access handling, thereby minimizing invasive changes to the existing route tree while keeping decisions deterministic and centrally auditable.

5.3.1 State model and eligibility rules

The incremental translation model is realized by introducing an explicit, discrete state space for each pair of (namespace, language). The state space is represented by a constant `TranslationStatus` with the values `missing`, `translated`, and `approved`. This directly reflects the architectural distinction from Section 4.4: translated denotes completeness (100% translated), while approved additionally captures a review threshold.

Route eligibility is computed by projecting namespace states to the route level via the route-to-namespace mapping already established in Section 5.2. Concretely, route-level predicates derive the required namespace set by matching a route name or path against the route tree and collecting `ns` declarations along the match chain, always including the baseline namespace 'common'. A language is treated as eligible for a route if all required namespaces are in state `translated` or `approved`. This predicate is implemented as `isRouteFullyTranslated({name, lang})`, which returns true for English unconditionally and otherwise checks the coverage map for every required namespace.

In addition to route availability, the system differentiates between "fully translated" and "auto-translated" routes in order to support the disclosure requirement from Section 4.4.4. The predicate `isRouteAutoTranslated({ routePath, lang })` classifies a route as auto-translated if at least one required namespace is in state `translated` (but not `approved`) for the effective language. This separates functional completeness (required for publication) from review completeness (required to suppress disclosure), while keeping both decisions derived from the same namespace-level state model.

Figure 5.12 shows the three translation states and the implementation of the

auto-translation classification predicate.

```
import translationCoverage from '../../../translation-coverage.json' with {
    type: 'json' };

export const TranslationStatus = {
    MISSING: 'missing',
    TRANSLATED: 'translated',
    APPROVED: 'approved',
};

export const isRouteAutoTranslated = ({ routePath, lang }) => {
    if (!lang || lang === 'en') return false;
    const nsList = getTranslationNamespacesForRoutePath(routePath);
    const nsCoverage = translationCoverage?.namespaces || {};
    return nsList.some((ns) => nsCoverage?.[ns]?.[lang] === TranslationStatus.
    TRANSLATED);
};
```

Figure 5.12: Implementation of translation state model and auto-translation classification predicate

5.3.2 Coverage file

The translation state model is backed by a coverage artifact stored as a JSON file `translation-coverage.json`. The file maps namespace identifiers to per-language state values. At runtime, `routes.mjs` imports this file as a module and treats it as the single source of truth for incremental translation decisions. The use of a JSON import provides a build-time resolved, deterministic view of coverage state, and it avoids introducing an additional network dependency for availability checks.

Operationally, the coverage file is updated through a build/automation task that consumes translation progress data per component/namespace from Weblate. This is further described in the section on CI integration (Section 5.4).

The coverage is expressed at namespace granularity, which aligns with the modular localization architecture and allows for fine-grained tracking of translation progress. Each namespace can independently transition through the three states for each language, enabling incremental rollout without requiring all namespaces to be completed simultaneously.

A small excerpt of the coverage file is shown in Figure 5.13.

```
{
    "namespaces": {
        "about": {
            "en": "approved",
            "de": "approved",
            "fr": "missing",
            "tr": "missing"
        },
        "common": {
            "en": "approved",
            "de": "approved",
            "fr": "translated",
            "tr": "translated"
        },
        "contact": {
            "en": "approved",
            "de": "approved",
            "fr": "missing",
            "tr": "missing"
        },
        ...
    }
}
```

Figure 5.13: Excerpt of translation-coverage.json

5.3.3 Publication-aware routing

Publication awareness is implemented at the URL generation boundary. The exported route helper `generatePath({ name, lang, ... })` extends the baseline path generation logic by applying the eligibility predicate before emitting a localized URL. If the requested language is not English and the route is not translated in that language, `generatePath` falls back to the default language variant and returns the corresponding URL without an /en prefix. This achieves two effects: it prevents creation of links to language-prefixed URLs for unpublished routes, and it ensures that fallback targets are canonical default-language URLs rather than secondary English aliases. The same publication-aware resolution is also used by the `LanguageSelector`. Rather than presenting all configured languages unconditionally, the selector only offers those languages for which the current route is actually published. As a result, users are not presented with language-switching options that would lead to fallback behavior or unavailable language variants.

Direct-access behavior for public pages is handled in the client-side minimal `publicPages.js` bundle through a canonical-driven redirect function. On initial page load, the script determines the desired language as either the language prefix present in the URL, or a previously stored language preference. It then consults `<link hreflang="...">` elements in the document head to obtain the canonical-equivalent path for that language, and redirects if the current path

differs. Because these hreflang targets are generated through the same path generation utilities as internal navigation, the redirect mechanism reuses the publication-aware resolution indirectly: for unpublished routes in a language, the corresponding hreflang entry is not present. In addition, direct accesses to /en/... are normalized by removing the /en prefix, maintaining a single canonical URL space for the default language.

This approach keeps publication gating centralized in route utilities and head metadata rather than duplicating availability logic in multiple places. Because the `hreflang` targets are generated from the same publication-aware path resolution used for routing and static generation, only published language variants appear as alternate links. The redirect mechanism therefore remains consistent with the publication model by construction. Since the public SSG output includes these metadata entries, direct-access redirection operates deterministically for the emitted pages without requiring a separate availability check in the client.

Figure 5.14 shows the implementation of the publication-aware `generatePath` function and the predicate to check if a route is fully translated for a given language.

```
1  const isRouteFullyTranslated = ({ name, lang }) => {
2      if (!lang || lang === 'en') return true;
3      const nsList = getTranslationNamespacesForRouteName(name);
4      const nsCoverage = translationCoverage?.namespaces || {};
5      return nsList.every((ns) =>
6          [TranslationStatus.APPROVED, TranslationStatus.TRANSLATED].includes(
       nsCoverage?.[ns]?.[lang])
7      );
8  };
9
10 export const generatePath = ({ name, lang, params = {}, explicit = false }) =>
        {
11     const getPath = ({ name, lang, params, explicit }) => {
12         const path = _generatePath(getPathPattern({ name, lang, explicit }),
       params);
13         if (path.endsWith('/') || path.endsWith('*')) return path;
14         else return `${path}/`;
15     };
16     const shouldFallback = lang !== 'en' && !isRouteFullyTranslated({ name,
       lang });
17     if (shouldFallback) {
18         return getPath({ name, lang: 'en', params, explicit: false });
19     }
20     return getPath({ name, lang, params, explicit });
21 };
```

Figure 5.14: Implementation of generatePath

5.3.4 Translation debug mode

The translation debug mode described in Section 4.4.3 is implemented as a small extension of the existing message generation and loading logic. Its purpose is to make unapproved translations visible in the running application without changing component code.

The first part is implemented within the language bundle generation function. Besides generating the normal namespace-based JSON catalogs, this function supports an optional parameter `debugTranslation`. If this flag is enabled, the generator queries Weblate for the approval state of the translation units belonging to the current namespace-language pair. For each message identifier, it checks whether the corresponding unit is approved. Unapproved messages are prefixed with a visual marker (a big red circle), while approved messages remain unchanged. The result is written as a debug variant of the normal message catalog to `<namespace>.<lang>.debug.json`. This file coexists with the normal translation file and is not used by default.

The second part is implemented in `loadMessages` already shown in Figure 5.10. Debug mode is activated through the query parameter `debugTranslation=true`. When present, this value is stored in `sessionStorage` so that the mode remains active during the current session. During namespace loading, the loader checks whether debug mode is enabled and requests either the normal `.json` files or the corresponding `.debug.json` files. One important detail is that in order for the debug mode to work, the CSR bundle must be served from the backend. This is achieved by filtering for the presence of the `debugTranslation` query parameter in the request and then serving the CSR application.

No separate rendering path is required. The runtime still consumes a regular message object, and the rest of the internationalization logic remains unchanged. Existing caching, request deduplication, and fallback to the default language are reused without modification.

Overall, the implementation is designed to minimize changes to the application code: approval information is incorporated during file generation, and runtime support is confined to selecting the debug artifact variant. This makes unapproved strings directly visible in application context while leaving the existing internationalization logic largely unchanged and keeping Weblate as the source of truth for the approval state.

5.3.5 Language-change link hinting

To reduce user confusion when navigation implies a language change, a custom link component integrates a lightweight language-change indicator consistent with Section 4.4.5. The implementation derives the current language from the URL

prefix (defaulting to English when no prefix exists), resolves the target path, and compares the target language inferred from the target pathname against the current URL language. If the languages differ, the component annotates the link with a small indicator (implemented as a globe symbol) and sets a tooltip.

Figure 5.15 shows an example of the language-change hinting in the rendered application.

Figure 5.15: Language-change hinting for cross-language links

The indicator is intentionally constrained to internal navigation. External targets (detected via URL schemes such as `https://`, `mailto:`, or `tel:`) are excluded. Additionally, the API exposes an option to suppress annotation where it would be redundant or visually disruptive (e.g., for language switchers or icon-only links).

An important interaction arises from publication-aware path generation. When a link target route is requested in the current language but falls back to the default language due to missing coverage, the resulting target pathname is non-prefixed and therefore inferred as English. Consequently, the link is treated as a cross-language navigation and receives the indicator and tooltip automatically. This ensures that fallback-driven language switches remain visible to users without requiring link authors to implement special-case logic.

The implementation of the custom `Link` component is provided in Appendix B.

5.3.6 Auto-translation disclaimer

Routes that are eligible but not sufficiently reviewed (`translated`) require explicit user-facing disclosure. This is implemented as a sticky notice bar component that is rendered at the top of affected pages. The component determines the current language from the URL prefix and classifies the current route via `isRouteAutoTranslated({ routePath, lang })`. The disclosure is therefore strictly tied to the same route-to-namespace and coverage map logic used for publication gating.

The notice provides three actions aligned with the architectural requirements. First, “Show original” navigates to the English equivalent of the current route. Second, the notice can be dismissed for the current session. This is persisted in session storage with a route-specific key, ensuring that dismissal does not globally

suppress disclosure across unrelated pages. Third, the disclosure can be disabled persistently for the current language. This is stored in local storage using a language-scoped key.

Figure 5.16 shows an example of the auto-translation disclaimer notice bar in the rendered application.

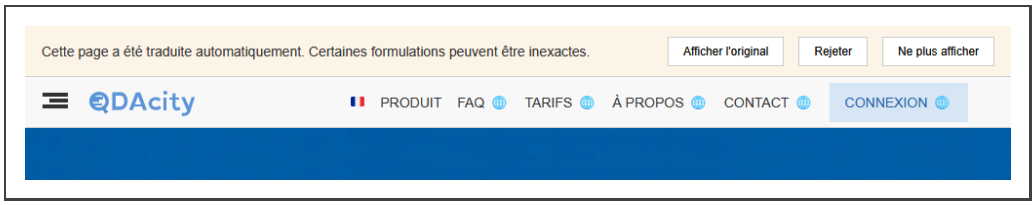

Figure 5.16: Auto-translation disclaimer notice bar

The implementation deliberately scopes persistence to (language, route) for session dismissal and to (language) for persistent suppression. This reflects the trade-off between transparency and fatigue: repeated disclosure on the same page within a session is avoided, while a user can still opt out of the notice entirely for a language once they accept the risk profile of auto-translated content.

The implementation of the `AutoTranslationDisclaimer` component is provided in Appendix C.

5.3.7 SEO coupling

Incremental translation affects discoverability and indexing, and the implementation therefore couples SEO metadata emission to the publication model. Two integration points are relevant.

First, build-time sitemap generation filters alternate language entries using the same route availability predicate. In the sitemap task, each public English URL is emitted as a `<url>` entry, and hreflang alternates are restricted to those languages for which `isRouteFullyTranslated({ name, lang })` returns true. This ensures that crawlers observe only published language variants in the sitemap and are not encouraged to discover language-prefixed URLs that would fall back to English.

Second, metadata for the `<head>` tag of the static HTML document is emitted through the helmet component. The component generates hreflang alternates for configured languages by calling the same path generation interface used throughout the application.

A notable coupling exists between SEO metadata and direct-access handling: the client-side redirect logic resolves the “correct” localized URL by reading hreflang

link elements from the head. This makes head metadata not only a crawler-facing artifact but also an input into runtime behavior, reducing duplication of route resolution logic.

This section completes the implementation view of incremental translation at runtime and build time. Section 5.4 extends the perspective to the translation lifecycle and CI automation that produces and updates the coverage artifact consumed here, thereby closing the loop between translation management and publication-aware navigation.

5.4 Weblate Integration and CI Automation

Section 5.3 relies on a translation status artifact to decide route availability per language and to trigger user-facing disclosure for unapproved translations. The artifact, however, is only reliable if translation updates are synchronized deterministically, reviewed in a controlled manner, and accompanied by an automated update of translation status metadata. This section describes the implemented Weblate integration and the corresponding GitLab CI automation that keeps English templates in sync with the translation platform, brings Weblate-originated translation updates back into the main codebase via structured merge requests, and produces translation status updates consumed by the runtime decision logic. The solution follows the CI requirements defined in Chapter 3, in particular FR-CI-03 (deterministic synchronization) and FR-CI-04 (structured merge requests), and operationalizes FR-LC-01 by maintaining a consistent translation status artifact.

5.4.1 Motivation and Constraints

The primary motivation for integrating Weblate through CI automation is to decouple translation work from feature development while still ensuring that incoming translation updates remain reviewable, reproducible, and consistent with the application's namespace contracts. In the implemented approach, Weblate serves as the translation management surface, but the repository remains the authoritative storage for translation artifacts consumed at runtime. This imposes two constraints.

First, English template changes must be propagated to Weblate in a controlled way. Since templates are derived from message extraction and namespace splitting, they cannot be edited manually without risking drift from the source code. The integration therefore treats template publication as an automated CI responsibility, based on build outputs rather than ad-hoc commits.

Second, translation updates originating from Weblate must re-enter the codebase through a workflow that preserves review and minimizes merge conflict risk. In

particular, Weblate commits can touch many namespaces and languages simultaneously (without partitioning). This yields large and difficult-to-review merge requests. The automation therefore enforces a partitioning policy that scopes merge requests by language, and optionally by individual namespace file when changes exceed a configured threshold (FR-CI-04). The resulting diffs are generated from Git comparisons and applied as patches onto clean target-branch checkouts, ensuring deterministic, auditable changes limited to translation artifacts and their associated metadata (FR-CI-03).

Beyond these technical goals, the integration is also motivated by the requirement that translation and review work can be performed by non-developers without requiring developer-specific tools. Translators and reviewers should not need to use Git, GitLab, or direct editing of text-based translation files. Instead, they can translate and approve strings within Weblate's user interface. This requirement introduces a workflow constraint: all necessary operations for translation progress (translation, review/approval, and discussion) must be achievable within Weblate, while synchronization to the repository and subsequent integration into the main branch is handled by automation. The architecture therefore treats Git/GitLab interaction as an implementation detail of the synchronization layer rather than as a prerequisite for translation work.

A further constraint stems from incremental translation. Because publication and disclosure decisions are driven by translation status per (language, namespace) pair, translation updates must be accompanied by status updates derived from Weblate's statistics. The integration therefore includes a build-time task that fetches translation and approval percentages per Weblate component and writes the derived state into the translation status artifact.

5.4.2 Branch strategy

The integration uses an explicit branch separation between the main development branch and a dedicated Weblate synchronization branch. English template updates are published to the Weblate branch, and incoming commits are confined to that branch. From there, CI creates new branches and opens merge requests that apply the Weblate branch's translation diffs onto the main branch as reviewable, scoped changesets.

This branch strategy provides two benefits. First, it prevents direct translation commits from landing on the main branch without review, while still enabling Weblate's Git-based workflow to operate normally. Second, it ensures that merges from the main branch into the Weblate branch can occur without affecting the integrity of the translation diffs that are later proposed for review: CI constructs merge requests by applying patches derived from Git comparisons rather than merging the whole branch.

In GitLab CI, this separation is reflected directly in job rules. Template synchronization is configured to run on the main branch, whereas merge request creation is configured to run only for pipelines triggered on the Weblate synchronization branch.

Figure 5.17 illustrates the interaction between the different branches and Weblate.

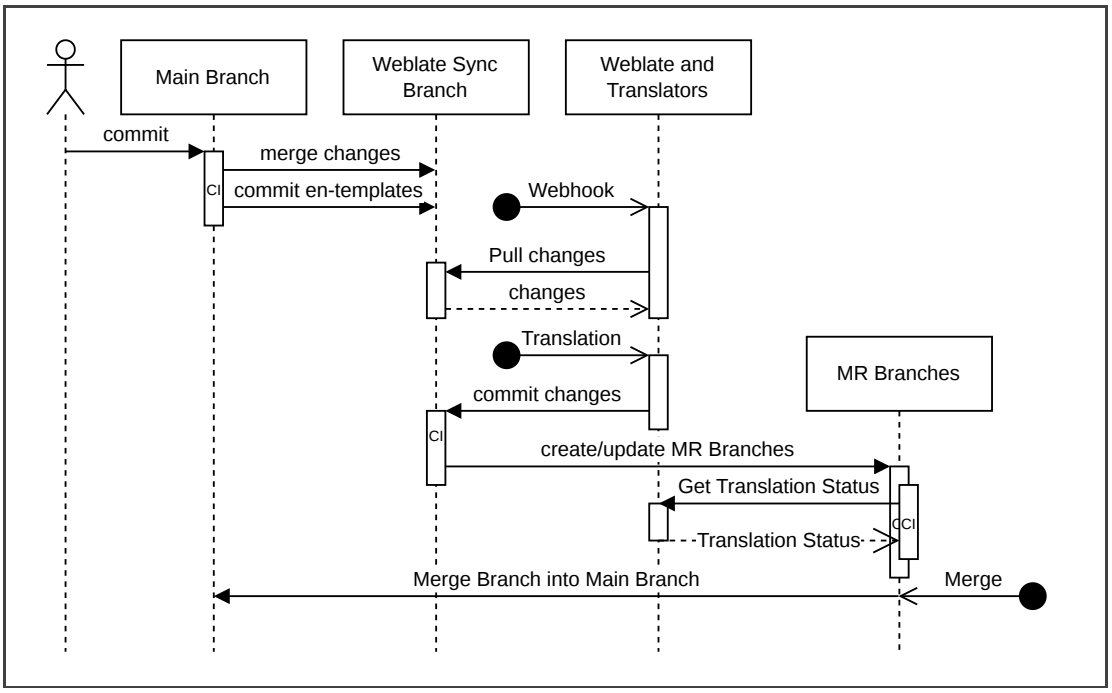

Figure 5.17: Sequence diagram illustrating branch dependencies (drawn with draw.io)

5.4.3 Syncing English templates to Weblate

English templates are generated as part of the production build output by splitting the extracted English message catalog into per-namespace template files `<namespace>.en.txt`. The CI build job persists these template artifacts so they can be reused by a downstream synchronization job. In the provided pipeline configuration, the frontend build exports the created `<namespace>.en.txt` files as an artifact, which establishes a stable interface between template generation and Weblate synchronization.

In addition to pushing template updates to the Weblate synchronization branch, the integration ensures that Weblate pulls new commits from GitLab promptly. Concretely, whenever a commit is made to the Weblate synchronization branch `weblate-translation`, a GitLab webhook is triggered that notifies Weblate to pull the latest repository state. This avoids reliance on periodic polling and re-

duces latency between template publication in GitLab and availability of updated source strings in Weblate.

Synchronization to the Weblate branch is implemented as a shell script that fetches the main branch and the Weblate branch, temporarily archives the generated templates, checks out the Weblate branch and merges the main branch into it, and replaces the Weblate branch's templates with the archived versions before committing and pushing the result. The commit is created with an explicit "skip CI" marker to avoid recursively triggering redundant pipelines on the synchronization branch.

This design keeps the template publication deterministic. Templates are never edited manually in the Weblate branch, they are always overwritten by generated artifacts, ensuring that the translation platform reflects the current message extraction result and the current namespace partitioning.

Figure 5.18 provides the implementation of the synchronization script used in the CI job.

```
# set default values for variables if not set in CI/CD settings
WEBLATE_BRANCH="${WEBLATE_BRANCH:-weblate-translation}"
EN_TEMPLATES_GLOB="${EN_TEMPLATES_GLOB:-war/translations/*.en.txt}"
MASTER_BRANCH="${CI_DEFAULT_BRANCH:-master}"

git config --global --add safe.directory "$CI_PROJECT_DIR"
git config user.name  "Master To Weblate Sync Bot"
git config user.email "master-to-weblate-sync-bot@example.invalid"

git fetch origin "$MASTER_BRANCH"
git fetch origin "$WEBLATE_BRANCH" || {
  echo "Branch '$WEBLATE_BRANCH' not found. Create it once in GitLab, then rerun."
  exit 1
}

tmp="$(mktemp -d)"
trap 'rm -rf "$tmp"' EXIT INT TERM

# Save current generated files before switching branches
tar -cf "$tmp/en_templates.tar" $EN_TEMPLATES_GLOB

: "${GITLAB_TOKEN:?Missing GITLAB_TOKEN CI variable (needs write_repository)}"
git remote set-url origin "https://oauth2:${GITLAB_TOKEN}@${CI_SERVER_HOST}/${CI_PROJECT_PATH}.git"

# Clean up any local changes
git reset --hard

# Checkout target branch
git checkout -B "$WEBLATE_BRANCH" "origin/$WEBLATE_BRANCH"

git merge -X theirs -m "Merge ${MASTER_BRANCH} into ${WEBLATE_BRANCH} [skip ci]" "origin/$MASTER_BRANCH" || {
  echo "Merge conflict. Resolve conflicts in '$WEBLATE_BRANCH' and rerun."
  exit 1
}

base_dir="${EN_TEMPLATES_GLOB%/*}"
rm -f "$base_dir"/*.en.txt 2>/dev/null || true
tar -xf "$tmp/en_templates.tar"

git add -f $EN_TEMPLATES_GLOB

# Commit template changes if any
if ! git diff --cached --quiet; then
  git commit -m "Sync English translation templates from ${MASTER_BRANCH} [skip ci]"
else
  echo "No template changes to commit."
fi

git push origin "$WEBLATE_BRANCH"
```

Figure 5.18: Script for syncing English templates to Weblate branch

5.4.4 Merging Weblate translation updates back to the codebase

Incoming translation updates are integrated back into the main branch through a dedicated CI job that runs on the Weblate synchronization branch. Instead of merging that branch into the main branch directly, the job constructs one or more scoped merge requests by computing diffs against the target branch and applying them as patches onto clean target-branch checkouts.

This procedure operationalizes the "structured merge request" requirement (FR-CI-04) by enforcing predictable partitioning and by making each change set auditable and reviewable. It further supports determinism (FR-CI-03) because the generated diffs are derived from Git comparisons and are applied without additional formatting steps.

The algorithm for this process is illustrated in Figure 5.19.

```
1  Input: targetBranch, sourceCommit, translationsGlob, lineThreshold
2
3  1. Determine baseCommit as the common ancestor of targetBranch and
       sourceCommit
4  2. changedFiles ← all changed translation files between baseCommit and
       sourceCommit matching translationsGlob
5  3. changedLanguages ← extract locales from changedFiles
6  4. Remove language "en" from changedLanguages
7  5. If changedLanguages is empty:
8      Exit successfully
9  6. For each language in changedLanguages:
10     totalChangedLines ← sum of added and deleted lines across all changed
       files of language
11     If totalChangedLines > lineThreshold:
12         partitions ← one partition per changed file / namespace
13     Else:
14         partitions ← one partition containing all changed files of the
       language
15         For each partition in partitions:
16             6.1 Create or reset a CI branch from the current state of
       targetBranch
17             6.2 Generate a patch from baseCommit..sourceCommit restricted to
       the files of the current partition
18             6.3 If the patch is empty: Skip this partition
19             6.4 Apply the patch onto the clean CI branch
20             6.5 Determine the namespaces affected by the partition
21             6.6 Update translation status metadata for the language and
       affected namespaces
22             6.7 Commit the result to the CI branch
23             6.8 Push the CI branch to the remote repository
24             6.9 If an open merge request for this CI branch exists:
25                     Reuse it
26                 Else:
27                     Create a new merge request targeting targetBranch
```

Figure 5.19: Algorithm for merging Weblate translation updates back to the codebase

5.4.5 Consuming translation status from Weblate

The runtime's incremental translation decisions depend on the translation status artifact `translation-coverage.json`. To keep this artifact consistent with the translation platform, the implementation includes a Gulp task that derives translation states from Weblate's per-component statistics.

The Weblate API integration is implemented as a small utility module that issues authenticated requests to the Weblate REST API using an API token and base URL provided via environment variables. It supports two core operations relevant to the workflow.

- **Component bootstrap**: A helper `ensureComponent` lists existing components in a Weblate project and creates missing ones. A dedicated Gulp task scans the local translation directory, extracts the set of namespaces, and calls `ensureComponent` for each. This allows new namespaces introduced by development to be registered in Weblate without manual UI configuration.
- **Translation status derivation**: A second utility function `getComponentTranslationState` retrieves translation and approval percentages per language for a given component. A Gulp task maps these values to the state model used throughout the thesis: if a component is fully translated (100% translated), it becomes translated or approved depending on whether the approved percentage reaches the configured threshold (at least 90%). Otherwise it is set to missing. The task is parameterized by `--components` and `--lang` and updates only the requested entries inside `translation-coverage.json`, preserving the remainder of the artifact unchanged.

The merge request automation integrates this status update directly into each translation Merge Request (MR). After applying translation file changes for a language (and, where applicable, a single namespace), the script computes the affected namespaces and invokes the Gulp status task, staging the updated coverage file into the same commit. This keeps the translation status artifact aligned with the set of translation changes under review and ensures that publication and disclosure behavior can be validated against the same merge request content.

5.4.6 GitLab MR comments synchronization to Weblate

To reduce feedback fragmentation between code review and translation tooling, the implementation provides an optional mechanism to synchronize GitLab merge request inline comments back into Weblate's unit-level comment system. This enables reviewers to provide feedback where translation work is performed, while still using GitLab merge requests as the integration gate for repository changes.

The synchronization logic is implemented as a Node.js script that is invoked via a

dedicated Gulp task. The algorithm operates on open merge requests associated with the current branch and proceeds as follows.

1. **MR discovery:** The script determines the current branch name and extracts the project identifier from the Git remote URL. Using the GitLab API, it searches for an open merge request whose source branch matches the current branch.
2. **Discussion extraction:** The script retrieves MR discussions and filters for inline notes. For each inline comment, it records the new path and the line range.
3. **Namespace and language inference:** Only comments on translation files following the convention `<namespace>.<lang>.txt` are considered. From the file name, the script extracts namespace and lang, which are later used to scope Weblate unit queries.
4. **Unit resolution from line ranges:** For each comment, the script reads the referenced translation file and extracts the translation keys contained in the commented line range by parsing the id=value format. These keys become the "units" to be resolved in Weblate.
5. **Weblate unit lookup and comment posting:** For each unit key, the script queries the Weblate units endpoint using a search expression constrained by unit context, language, and project. If a matching unit is found, the script posts a comment to the unit's comment endpoint, prefixing the GitLab author name to preserve attribution. After posting, it sets the status of the corresponding unit to `needs editing`.

5.4.7 Post-release: IndexNow submission

As discussed in Section 4.3.7, sitemap generation is treated as a deterministic build artifact, while sitemap submission is treated as a release automation concern (FR-INC-08). The packaging step produces the verification material required for IndexNow by writing a key file into the distribution directory whenever `INDEX_NOW_KEY` is present. The key file is named `<INDEX_NOW_KEY>.txt` and contains the key itself, enabling IndexNow verification through static hosting of the file at the site root.

The submission step itself is performed post-release, after deployment of the updated sitemap and the corresponding verification key file. It is implemented as a Gulp task that submits all URLs changed by the last release to the IndexNow API. The identification of changed URLs is based on the `lastmod` field of the sitemap entries. If the `lastmod` matches the current day, the corresponding URLs are included in the submission batch. This ensures that only URLs affected by

the current release are submitted, avoiding redundant submissions of unchanged URLs.

The implementation of the submission task is provided in Figure 5.20.

```
export async function submitToIndexNow(urls) {
    const indexNowKey = process.env.INDEX_NOW_KEY;
    const indexNowHost = 'https://www.bing.com';

    if (!indexNowKey) {
        console.debug('No INDEX_NOW_KEY provided. Skipping submission to IndexNow.');
        return;
    }

    const payload = {
        host: 'www.qdacity.com',
        key: indexNowKey,
        keyLocation: `https://www.qdacity.com/${indexNowKey}.txt`,
        urlList: urls,
    };

    const response = await fetch(`${indexNowHost}/indexNow`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(payload),
    });
    if (response.ok) {
        console.debug('Successfully submitted URLs to IndexNow.');
    } else {
        throw new Error(
            `Failed to submit URLs to IndexNow. Status: ${response.status}, Response: ${await response.text()}`
        );
    }
}
```

Figure 5.20: Gulp task for submitting changed URLs to IndexNow

5.5 DeepL Integration for Developer Tooling

The Weblate-based workflow described in Section 5.4 is primarily designed for translators and reviewers. Machine translation support is nevertheless useful earlier in the lifecycle, when developers introduce new message identifiers or refactor existing strings. In this phase, translation files are often incomplete by construction, which can either block verification steps or produce under-translated languages until the next translation cycle. To reduce this friction, the implementation provides a developer-oriented DeepL integration that can detect missing, removed, or changed messages at namespace granularity and pre-fill affected translation entries with machine translations that are explicitly marked for sub-

sequent review. This aligns with the architecture's notion of machine translation as an accelerator rather than a replacement for review and approval.

5.5.1 Detecting new, changed or deleted messages

Detecting missing and extra identifiers For each translation file, the implementation derives the namespace slug and language code from the file name and converts the slug into the camelCase prefix used by message identifiers (e.g., landing-page → landingPage). The English template is then filtered to the effective namespace by selecting only IDs that start with `<namespaceCamel>`. The resulting set of message IDs represents the expected identifiers for the translation file, and the parser-derived identifier list represents the actual identifiers currently present. Missing and extra entries are computed by comparing these two sets:

- missingInTranslation: identifiers that exist in the English template but not in the translation file
- extraInTranslation: identifiers that exist in the translation file but not in the English template

Detecting changed English source strings In addition to missing and removed identifiers, the tooling can detect when the English source text for an existing identifier changed. This detection is limited to the current developer workspace and targets uncommitted changes. The implementation identifies staged and unstaged source files, extracts messages from the working-tree versions of those files, and compares them to messages extracted from the corresponding `HEAD` versions of the same files. If the extracted defaultMessage differs, the identifier is treated as changed. This produces a list of changed IDs that can be intersected with a translation file's identifier set to identify entries that require re-translation.

The implementation of this detection logic is provided in Figure 5.21.

```
const computeChangedEnglishMessageIdsUncommitted = async (currentEnglishJson)
    => {
    const changedFiles = listUncommittedChangedSourceFiles();
    if (!changedFiles.length) return [];
    const currentFromFiles = await extractEnglishFromFiles(changedFiles);

    const headTmp = writeHeadVersionsToTemp(changedFiles);
    const repoRoot = runGit(['rev-parse', '--show-toplevel']);

    const headFiles = changedFiles
        .map((abs) => path.join(headTmp, path.relative(repoRoot, abs)))
        .filter((p) => fs.existsSync(p));

    const headFromFiles = await extractEnglishFromFiles(headFiles);
    fs.rmSync(headTmp, { recursive: true, force: true });

    const changedIds = [];
    for (const [id] of Object.entries(currentEnglishJson || {})) {
        const cur = currentFromFiles[id];
        const base = headFromFiles[id];
        if (!cur && !base) continue;
        const curMsg = cur?.defaultMessage;
        const baseMsg = base?.defaultMessage;
        if (curMsg !== baseMsg) changedIds.push(id);
    }
    return changedIds;
};
```

Figure 5.21: Implementation of uncommitted English source change detection

5.5.2 DeepL Configuration and API interaction

The DeepL integration is implemented as a small utility that translates batches of source texts for a target language. It is designed to be safe with respect to message formatting conventions used in React Intl, in particular ICU placeholders, and to support category-specific instruction hints for more consistent output.

Requests are authenticated via an API key provided through an environment variable. If the key is absent, the translation helper fails fast to avoid silent partial updates. Requests are sent to DeepL's translation endpoint.

Placeholder preservation and XML tag handling. To prevent placeholder corruption, the translation helper protects ICU placeholder blocks by replacing {...} segments with synthetic XML tags of the form `<x id="n"/>`. The request is configured with XML tag handling and a rule to ignore the `<x>` tag during translation. Before transmission, XML special characters are escaped in text segments while preserving tag structure. After translation, the `<x id="n"/>` markers are replaced with the original placeholder strings, and XML escaping is reverted in non-tag segments. This transformation is intentionally conservative: it preserves structural tokens and avoids accidental translation or reordering of

placeholders, while still allowing natural language translation of surrounding text.

Figure 5.22 provides the implementation of the ICU placeholder protection and unprotection logic.

```
function protectIcu(message) {
    const placeholders = [];
    let out = '';

    for (let i = 0; i < message.length; i++) {
        if (message[i] === '{') {
            let depth = 1;
            let j = i + 1;
            while (j < message.length && depth > 0) {
                if (message[j] === '{') depth++;
                else if (message[j] === '}') depth--;
                j++;
            }
            if (depth === 0) {
                const token = message.slice(i, j);
                const idx = placeholders.push(token) - 1;
                out += `<x id="${idx}"/>`;
                i = j - 1;
                continue;
            }
        }
        out += message[i];
    }

    out = escapeXmlPreservingTags(out);

    return { text: out, placeholders };
}

function unprotectIcu(translated, placeholders) {
    return unescapeXmlPreservingTags(translated).replace(/<x id="(\d+)"\/>/g,
    (_, n) => placeholders[n]);
}
```

Figure 5.22: Implementation of ICU placeholder protection and unprotection logic

Category-specific translation instructions. DeepL requests optionally include `custom_instructions` derived from the namespace being translated. The implementation defines a small set of text categories and assigns namespaces to categories through explicit rules. Categories include ”User Interface Elements”, ”Method Help”, and ”Product Pages”, each associated with short instruction lists. The effective instruction list is selected by matching the current namespace to the rule set.

This design is intentionally explicit rather than heuristic. Namespace-based categorization keeps the configuration auditable and stable over time, and it aligns with the route-bound namespacing. The task signals a warning if a namespace

does not match any category, so missing namespaces in this configuration are visible and can be added as needed.

Figure 5.23 provides the implementation of the category-specific instruction configuration.

```
const DEEPL_TEXT_CATEGORIES = Object.freeze({
    GENERAL: 'General Instructions',
    UI: 'User Interface Elements',
    METHOD_HELP: 'Method Help',
    PRODUCT: 'Product Pages',
});

const CATEGORY_RULES = [
    {
        category: DEEPL_TEXT_CATEGORIES.UI,
        namespaces: ['common', 'core'],
    },
    {
        category: DEEPL_TEXT_CATEGORIES.METHOD_HELP,
        namespaces: [
            'attentionToNegativeCases',
            'auditTrail',
            ...
        ],
    },
    ...
];

const CATEGORY_CONFIG = Object.freeze({
    [DEEPL_TEXT_CATEGORIES.GENERAL]: {
        instructions: [
            'While maintaining a professional tone of voice, when addressing the reader directly use the personal/informal form',
            ...
        ],
    },
    [DEEPL_TEXT_CATEGORIES.UI]: {
        instructions: ['Keep UI strings concise.', 'Prefer short, action-oriented wording.'],
    },
    ...
});
```

Figure 5.23: Implementation of category-specific DeepL instructions

5.5.3 Output formatting and developer workflow integration

DeepL translation is integrated as an optional mode of an existing "update translations" task. The task consumes the extracted English template `en.json` and applies repairs to the text-based translation files. This preserves the repository's translation file format and structure while allowing developers to generate consistent diffs after modifying message identifiers or default messages.

The update logic supports two distinct modes.

- Hint-only mode. If machine translation is disabled, missing identifiers are appended as commented "add" hints containing the English default message, and extra identifiers are annotated as "del" and commented out. This mode is designed to be non-destructive: it preserves existing translations and surfaces required edits without performing them automatically.
- DeepL-enabled mode. If machine translation is enabled, the update becomes constructive and applies updates to the translation files:
  - extra identifiers are removed from the translation file to keep it consistent with the namespace template.
  - missing identifiers are added with machine-translated values.
  - identifiers whose English source text changed are re-translated and updated in place.

Machine-generated entries are explicitly marked. When missing identifiers are translated and appended, the first inserted entry includes a comment block indicating that the following strings were automatically translated by DeepL and must be reviewed for accuracy. For updated existing entries, an additional note is appended to the description field to indicate that the update was performed automatically and requires review.

Developer workflow placement. The intended workflow is to run the update task after introducing new message identifiers or modifying default messages, before committing. In a typical sequence:

- Modify source code by introducing new message IDs or changing existing default messages.
- Run the translation update task in DeepL-enabled mode to populate missing entries and refresh translations whose English source changed.
- Review the resulting changes and remove the machine translation comments.
- Commit the resulting translation file changes to the repository, from which they can enter the Weblate-based review and approval workflow described in Section 5.4.

The key design decision is that DeepL is applied as a developer-side patch generator rather than a runtime mechanism. It improves iteration speed and keeps translation artifacts structurally consistent, while preserving the governance and transparency model enforced by Weblate, translation states, and the UI disclosure behavior.

5.6 Summary

Chapter 5 implements the proposed architecture as a production-ready delivery pipeline. It introduces a deterministic SSG process with parallel route rendering and a build-time sitemap generator derived from the route tree. The sitemap's lastmod is computed via static HTML body hashing with baseline-aware initialization to avoid falsely signaling "fresh" updates.

For localization, the chapter realizes route-bound namespaces with on-demand bundle loading, caching, request deduplication, and predictive prefetching to reduce navigation latency while keeping behavior consistent across CSR and SSG.

Finally, it operationalizes the translation workflow through CI/Weblate integration: structured translation merge requests are created automatically and include an updated translation coverage artifact, ensuring publication and disclosure logic can be reviewed against the same change set. DeepL-based tooling complements this with placeholder-safe machine translation and category-specific instructions for consistent output.

6 Evaluation

This chapter evaluates the implemented approach against the requirements defined in Chapter 3 and against the baseline system introduced in Chapter 4.2. In line with the requirements-driven framing of this thesis, the evaluation distinguishes between two forms of evidence.

First, a substantial part of the requirements is already addressed by the architectural design and by the concrete implementation artifacts documented in Chapters 4 and 5. Repeating these points in full would add redundancy rather than new insight. Therefore, this chapter begins with a compact overview of requirements whose fulfillment is already established structurally through the presented architecture and implementation. Where a requirement is inherently structural (e.g., explicit route-to-namespace binding, publication-aware path generation, or CI-based synchronization logic), the relevant evidence is therefore taken from Chapters 4 and 5.

Second, the central purpose of this chapter is the empirical and comparative evaluation of those requirements that cannot be justified by design alone, but require measurement, scenario-based validation, or critical assessment. The empirical comparison evaluates the baseline crawler-based prerendering approach against the target Node.js-based SSG under identical route and translation coverage. Where performance is assessed, the benchmark protocol defined in Section 3.5.1 is applied in its concrete form: repeated runs under the same environment, with a clean workspace, constant inputs, and reporting of median values and standard deviations. Accordingly, the remainder of the chapter focuses on build-time performance, parallel scalability, runtime behavior, translation-loading payloads, and the remaining correctness and maintainability aspects not already covered by the preceding chapters.

6.1 Requirements Already Addressed by Architecture and Implementation

Table 6.1 and Table 6.2 summarize the requirements whose realization is already established by the architecture and implementation chapters. These requirements are not excluded from validation altogether, but their essential solution mechanisms have already been documented and justified in the earlier chapters. The evaluation therefore only revisits them where additional evidence is necessary.

ID	**Requirement Name**	**Primary references**
NFR-07	Resilient failure handling	4.4.2, 5.2.4, 5.3.3
NFR-08	Route-to-namespace maintainability	4.3.2, 5.2.1
NFR-09	Sustainable toolchain	5.1
NFR-12	Language-consistent URLs	4.4.2, 5.3.3
NFR-13	Crawlable static output for eligible routes	4.3.3, 4.3.7, 5.1.4, 5.3.7
NFR-14	Compatibility with existing build toolchain and CI environment	4.1, 5.1, 5.4
NFR-15	Non-disruptive and accessible quality-state disclosure	4.4.4, 5.3.6

Table 6.1: Non-Functional Requirements whose realization is already established by Chapters 4 and 5

ID	Requirement Name	Primary references
FR-SSG-01	Configurable hybrid rendering	4.3.3, 5.1.4
FR-SSG-02	Language variants for eligible routes	4.3.3, 4.4.2, 5.1.4
FR-SSG-03	Deterministic server-side rendering environment	5.1.2, 5.2.3
FR-SSG-04	Module and asset resolution support	5.1.3
FR-SSG-05	Render isolation	5.1.4
FR-I18N-01	Namespaced message bundles	4.3.4, 5.2.1, 5.2.2
FR-I18N-02	Route-driven namespace resolution activation	4.3.4, 5.2.1, 5.2.3
FR-I18N-03	Caching and deduplication	4.3.4, 5.2.4
FR-I18N-04	SSG-compatible message loading	4.3.3, 5.2.3
FR-I18N-05	Bundle integrity and schema validation	5.2.1, 5.2.2
FR-I18N-06	Preload for initial render	4.3.4, 5.2.4
FR-I18N-07	Prefetch strategy	4.3.8, 5.2.5
FR-INC-01	Incremental rollout without mixed-language UI	4.4.1–4.4.2, 5.3.1–5.3.3
FR-INC-02	Deterministic navigation under missing translations	4.4.2, 5.3.3
FR-INC-03	Explicit indication of fallback-driven language change	4.4.5, 5.3.5
FR-INC-04	Route-specific language eligibility in language selection	4.4.2, 5.3.1, 5.3.3
FR-INC-05	Synchronized language selector	4.4.2, 5.3.1, 5.3.3
FR-INC-06	Direct access handling for unavailable language URLs	4.4.2, 5.3.3
FR-INC-07	User disclosure for unapproved translations	4.4.4, 5.3.6
FR-INC-08	Sitemap completeness	4.3.7, 5.1.6, 5.3.7
FR-INC-09	lastmod for published pages	4.3.7, 5.1.6
FR-INC-10	IndexNow submission for changed pages	4.3.7, 5.4.7
FR-LC-01	Translation status artifact consumption	5.3.1, 5.3.2, 5.4.5
FR-LC-02	Check validity of translation status artifact	5.4.5
FR-LC-03	Default-language change propagation for existing translations	5.5.1, 5.5.3
FR-LC-04	Machine-translation fill for newly introduced identifiers	4.3.5, 5.5.1, 5.5.2, 5.5.3
FR-LC-05	View for unreviewed translations	4.4.3, 5.3.4
FR-LC-06	Context visibility for unapproved translations	4.4.3, 5.3.4
FR-CI-01	Message identifier convention enforcement	5.2.1, 5.4
FR-CI-02	Namespace dependency and boundary enforcement	4.3.2, 5.2.1
FR-CI-03	Auditable and deterministic translation synchronization	4.3.5, 5.4.3, 5.4.4
FR-CI-04	Structured Merge Requests	4.3.5, 5.4.2, 5.4.4

Table 6.2: Functional Requirements whose realization is already established by Chapters 4 and 5

The requirements listed in Table 6.1 and Table 6.2 do not require dedicated fur-

ther empirical validation, because their essential solution mechanisms are already established by the architecture and implementation chapters. The main contribution of the present chapter therefore lies in the quantitative comparison, runtime inspection, and critical assessment of the remaining requirements that are not sufficiently addressed by the implementation description alone.

6.2 Requirements Not Implemented or Not Fully Met

Within the scope of the implemented solution, not all requirements were realized to the same degree.

FR-I18N-08 (Data saving option). The specified data-saving mode was not implemented. The current runtime supports route-bounded loading, caching, and predictive prefetching, but it does not provide a dedicated user- or system-controlled mode that disables speculative prefetching and restricts loading to the baseline namespace plus the currently active route only. Consequently, this requirement is not met.

FR-SSG-06 (Incremental regeneration). The thesis defines incremental regeneration as a desirable extension of the SSG. However, the implemented pipeline focuses on deterministic full-route rendering, worker-based parallelization, and publication-aware output generation. An execution mode that re-renders only changed routes based on incremental change detection is not implemented. Therefore, this requirement is not realized in the current scope.

NFR-10 (Runtime telemetry for translation loading). The maintainability goal of structured runtime telemetry is only partially met. The implementation contains warning-style logging and explicit failure handling in the translation-loading path, but it does not yet provide the fully structured telemetry described in the requirement, i.e., consistently emitted route-, language-, namespace-, latency-, and cache-state-attributed events. This gap does not invalidate the runtime architecture, but it limits analyzability under production-like conditions.

These omissions do not affect the core claim of the thesis, but they delimit the degree to which all requirements can be claimed as fully satisfied.

6.3 Empirical Evaluation

The raw measurements evaluated in this section can be found in Appendix D.

6.3.1 Environment and Tooling

All measurements are executed on a single local machine with the following characteristics:

- **CPU:** AMD Ryzen 7 7700X (8 cores / 16 logical processors, 4.50 GHz)
- **RAM:** 64 GB
- **Storage:** Samsung SSD 990 EVO Plus (2 TB)
- **Operating system:** Windows 11 Home
- **Node.js:** v22.21.0
- **npm:** 11.6.2

Build-time durations are recorded as reported by the corresponding Gulp build tasks. Runtime performance is measured using Chrome DevTools on Google Chrome 145.0.7632.117 (64-bit). LCP is collected under synthetic, controlled conditions using DevTools network throttling presets on the Desktop device profile.

6.3.2 Build-Time Performance and Parallel Scalability

This section evaluates the build-time performance of the prerendering pipeline and its scalability with respect to worker parallelism. The analysis is aligned with NFR-01 (reduced SSG build time) and NFR-02 (parallel scalability), and reports distribution-aware statistics (median and standard deviation) under Protocol P1.

Baseline vs. target pipeline. Table 6.3 summarizes prerender durations for the baseline (crawler-based) variant and the target SSG variant at different worker counts. The baseline exhibits a median prerender duration of 193.20 s (3.22 min), while the target pipeline reduces median duration substantially for all tested worker counts. The best median duration is achieved at 4 workers (6.99 s), corresponding to a 27.6× speedup over the baseline median. The standard deviation remains low across all configurations, indicating consistent performance across repeated runs.

Variant	n	Median [s]	σ[s]	Speedup vs. baseline (median)
Baseline (React-Snap)	10	193.20	0.01	–
Target (1 worker)	10	12.00	0.45	16.1×
Target (2 workers)	10	9.83	0.39	19.6×
Target (4 workers)	10	6.99	0.23	27.6×
Target (8 workers)	10	9.86	0.11	19.6×

Table 6.3: Prerender duration under Protocol P1 (across 10 repeated runs)

Scaling behavior. Increasing the worker count from 1 to 4 improves throughput (12.00 s → 6.99 s median). At 8 workers, median time increases again (9.86 s), indicating that overheads (e.g., worker setup, coordination, or shared I/O) outweigh further parallelization benefits for the given route corpus and environment. This satisfies the NFR-02 intent of demonstrating a worker configuration that improves over the single-worker setup, while also highlighting a practical upper bound where additional workers no longer improve time behavior.

HTML minification. In addition to prerendering, the target pipeline executes an HTML minification step. Measured independently, this step has a median duration of 6.93 s across the recorded runs. This step is run after the prerendering phase. However, as it is part of the HTML generation process, its duration is excluded from the overall prerender duration reported in Table 6.3 for the target pipeline, but is relevant for understanding the total build time and the contribution of different pipeline stages.

From a practical perspective, this result is central to the thesis contribution. The target pipeline does not merely replace an unmaintained dependency with an equivalent custom solution, but it improves the operational characteristics of the build in a way that directly benefits CI feedback cycles and route-scale growth.

6.3.3 Runtime Performance on Public Pages

Runtime performance was evaluated primarily through LCP on representative public routes, addressing NFR-03. The goal of this evaluation was not to show that the SSG replacement alone guarantees large runtime improvements in all network conditions, but rather to verify whether the target pipeline preserves or improves user-centric loading behavior on the evaluated public pages.

Method. LCP was measured for the fixed route subset. Measurements were taken under two DevTools network throttling presets (Fast 4G and 3G). For each configuration, the reported value is the 75^{th} percentile across the 10 repeated runs captured in the measurement dataset.

Results under Fast 4G. Table 6.4 summarizes the 75^{th} percentile for the selected routes. Across all measured routes, both variants achieve very low LCP values (all p75 values $\leq 0.31\,s$). Differences between baseline and target are small (on the order of $\pm 0.01\,s$) and are within the margin of error. For all evaluated routes and technologies, the standard deviations of the measurements were below 0.01 s, indicating very consistent performance.

Route	Baseline p75 [s]	Target p75 [s]	Δ [s]
/	0.30	0.30	0.00
/de/	0.30	0.31	0.01
/about/	0.29	0.28	-0.01
/de/ueber-uns/	0.29	0.29	0.00
/qda-software/	0.30	0.31	0.01
/de/qda-software/	0.30	0.30	0.00

Table 6.4: LCP (p75) under Fast 4G throttling for selected public routes

Results under 3G. Table 6.5 summarizes the p75 LCP values under 3G throttling. In contrast to Fast 4G, all measured p75 values are above the "good" threshold of 2.5 s (Table 2.1), placing the tested pages into the "needs improvement" range under this synthetic constraint. Compared to the baseline, the target pipeline shows slightly higher LCP values across the measured routes (+0.05 to +0.10 s at p75). Given that the experiment isolates the SSG tooling (with unchanged application assets), these differences are small relative to the overall 3G-induced latency and should be interpreted as a minor regression rather than a structural shift in runtime behavior. Under 3G throttling as well, the standard deviations of the measurements were below 0.03 s.

Route	Baseline p75 [s]	Target p75 [s]	Δ [s]
/	3.50	3.55	0.05
/de/	3.43	3.50	0.07
/about/	3.46	3.53	0.07
/de/ueber-uns/	3.42	3.48	0.06
/qda-software/	3.47	3.57	0.10
/de/qda-software/	3.40	3.47	0.07

Table 6.5: LCP (p75) under 3G throttling for selected public routes

Interpretation. Overall, the results indicate that replacing crawler-based prerendering with the target SSG does not materially change LCP for the evaluated public pages under the tested conditions. Under Fast 4G, both variants already achieve values well within the "good" range. Under 3G, both variants exceed the 2.5 s threshold, suggesting that network constraints dominate the critical path in this synthetic setup. One plausible explanation for the slight regression is that browser-based snapshotting is already near the optimum achievable for these pages, because it captures only the DOM state actually rendered in the browser. The target SSG, by contrast, produces a slightly larger HTML resource, partly due to conservative pruning of header contributions such as Font Awesome, so the small LCP increase is attributable to this minor HTML overhead rather than to a substantive change in runtime behavior.

The performance evaluation therefore yields a nuanced result. The target approach does *not* demonstrate a clear LCP advantage over the baseline on the sampled public pages. However, it also does not introduce a meaningful runtime degradation under the more relevant Fast 4G condition, while simultaneously delivering major build-time benefits and improved control over publication, payload, and translation behavior. In that sense, the requirement is only partially satisfied: the implementation remains within a comparable runtime performance range on the evaluated routes, but it does not consistently outperform the baseline for LCP.

This does not exclude runtime benefits in other parts of the system: for the authenticated CSR application shell, where translation payload is part of the critical rendering path, the target architecture showed a much clearer improvement, as discussed in Section 6.3.4.

6.3.4 Translation Loading Payloads and Redundant Downloads

Another quantitative focus concerns the runtime cost of translation loading. This addresses NFR-04 (route-bounded translation payload).

The architecture and implementation already establish that translation loading is scoped to the shared baseline namespace and the namespaces declared by the active route. The evaluation complements this design argument with runtime observation. Under the route-aligned namespace model, namespace artifacts were small (approximately 10–15 kB per namespace in the evaluated scenarios), which bounded the cost of predictive prefetching and limited the additional transfer overhead when route-specific messages were fetched ahead of navigation.

The qualitative behavior of the loader was consistent with the design goals. Caching and request deduplication ensured that once a namespace had been loaded, subsequent accesses did not trigger redundant network transfers within the same session. Likewise, prefetch requests reused the same cache and in-flight request tracking as normal navigation-triggered loading, preventing duplicate downloads. This is especially important because speculative loading only remains attractive if it does not multiply transfer cost unnecessarily.

CSR application startup payload. The effect of route-aligned namespace splitting becomes more visible in the authenticated application than on the already fast public pages. In QDAcity, the public pages were split into page-specific namespaces, whereas the authenticated application uses a shared `core` namespace that contains the texts required across the main application area. In the baseline architecture, this startup-relevant translation resource was effectively part of a monolithic per-language catalog of 1319 kB. In the target architecture,

the initially required translation resource for the authenticated application was reduced to 260 kB, corresponding to a reduction of approximately 80.3%. Since these translation resources were delivered with gzip compression, the effective transfer size was smaller in both cases: 288 kB in the baseline and 44.75 kB in the target architecture. This shows that the route-aligned namespace model not only reduces the nominal size of the startup-relevant translation resource, but also substantially lowers the amount of data actually transferred during application startup.

Because this translation resource lies on the critical path of the authenticated application's initial rendering, the payload reduction also translated into clearly improved loading behavior. Under Chrome DevTools Fast 4G throttling, repeated measurements consistently showed lower LCP values for the target architecture, with the median decreasing from 3.11 s in the baseline to 2.09 s in the target architecture. This indicates that, even where public-page LCP remained broadly comparable, the namespace-based reduction of startup translation payload provides a substantial runtime benefit for the authenticated CSR application shell.

Variant	Initial translation resource size	Gzipped transfer size	Median LCP [s]
Baseline	1319 kB	288 kB	3.11
Target	260 kB	44.75 kB	2.09

Table 6.6: Initial translation payload, gzipped size, and median LCP for the authenticated application

Compared to the monolithic baseline, the route-aligned namespace model changes the scaling characteristic of translation payloads: the transfer cost depends on what the user visits rather than on the total size of the translated application. This is one of the clearest confirmations of the thesis rationale. Even where the runtime LCP gain is small, the runtime translation-loading model becomes more precisely bounded and therefore more scalable as the application and language set grow.

6.3.5 Prefetching and Navigation Behavior

The hover-based prefetch mechanism was evaluated not primarily as a raw speed optimization, but as a navigation-stability optimization under slower network conditions. On fast connections, the visible difference between navigation with and without prefetching was negligible. Under throttled conditions, however, prefetching reduced brief flashes of fallback or default-language strings before localized route content became fully available.

This result is important because it shows that the implemented prefetch strategy improves perceived continuity rather than headline benchmark numbers. The effect is bounded: prefetching does not transform overall page performance, and the current implementation relies on hover as an intent signal, which is more appropriate for pointer-based devices than for touch-based contexts. Still, within its intended scope, the mechanism supports the thesis objective of reducing route-transition friction without abandoning lazy loading.

Accordingly, FR-I18N-07 is operationalized successfully, but the evaluation also shows why it should be interpreted as an optimization rather than as a primary correctness mechanism. The system remains correct without prefetching. The benefit of prefetching is that it smooths transitions under adverse network conditions with limited bandwidth overhead.

6.3.6 Correctness, URL Consistency, and Cross-Mode Behavior

Beyond performance, the evaluation also examined the correctness of the publication-aware localization model and its consistency across SSG and CSR delivery modes. This concerns, in particular, FR-INC-01, FR-INC-02, FR-INC-06, FR-INC-07, NFR-06, and NFR-12.

Scenario-based inspection showed that the publication-gated model behaved as intended for the evaluated routes. Language-specific routes were only linked and published when the required namespaces reached the required translation state. If a route was unavailable in the requested language, navigation resolved deterministically to the default-language canonical route rather than generating a broken or misleading language-specific URL. Direct access to unavailable localized URLs was likewise normalized consistently to the default-language canonical form.

The evaluation also compared the user-visible behavior of routes first entered as SSG pages with subsequent CSR navigation under incomplete translations and delayed namespace loading. Within the evaluated route subset, the same publication rules and quality-state logic remained visible in both delivery modes. This supports the thesis claim that the route-centric localization model does not depend on environment-specific switches, but instead uses the same availability logic across static output, runtime navigation, and crawler-facing artifacts.

The evaluation further confirmed two additional correctness properties of the SSG output. First, repeated SSG runs with identical inputs produced identical HTML artifacts for the evaluated routes, showing that the emitted files remained deterministic under the conducted tests. This satisfies NFR-05 for the evaluated scope. Second, the generated pages hydrated correctly in the browser, both for pages using the minimal `publicPages.js` bundle and for pages using the

full React bundle. When the emitted static artifacts were served and loaded with developer tools open, no hydration mismatch warnings or hydration-related runtime errors were observed, and the pages became interactive as expected after load. This satisfies FR-SSG-07 for the evaluated route-language variants covered by the tests.

For the evaluated scope, the strongest result here is not a single numeric metric but the absence of contradictory behavior across delivery modes: the URL, published route set, and disclosure behavior remained aligned with the translation state model. This is a crucial correctness property because it ensures that incremental localization does not degrade into mixed-language or misleadingly localized route variants.

6.3.7 Maintainability and Workflow Evaluation

Maintainability is evaluated here in the narrower sense relevant to this thesis: whether the new approach improves analyzability, modifiability, and governance compared to the baseline. This concerns NFR-09, NFR-11, and the associated lifecycle and CI requirements.

The most important maintainability result is conceptual but still evaluative: the target solution replaces implicit crawling behavior and monolithic translation artifacts with explicit contracts at stable architectural boundaries. Route-level namespace declarations make translation dependencies explicit and reviewable in the route configuration. CI checks enforce identifier conventions and namespace usage rules. Translation updates re-enter the repository through deterministic and partitioned merge requests. Furthermore, the SSG process itself is based on an owned pipeline rather than on an externally unmaintained crawler.

The requirement for developer documentation is addressed through a combination of structured and, where necessary, documented code and dedicated internal documentation. Where further explanation is required, the implementation is supported by five pages in the application's internal wiki. These cover the namespace and localization architecture with the associated CI checks and automation scripts, the initialization of Weblate for the translation project, guides for introducing new languages or namespaces, and the DeepL integration for machine-supported translation.

6.4 Limitations

The results presented in this chapter should be interpreted in the context of the implemented scope and the chosen evaluation setup. The purpose of the evaluation was to compare the baseline and the target solution within the QDAcity environment and to assess whether the main engineering objectives were achieved.

It was not intended to derive universally applicable performance figures or to cover all possible usage contexts.

Evaluation context. The reported measurements were obtained under a controlled local setup. While this supports a consistent comparison between the baseline and the implemented solution, the results remain influenced by the characteristics of the underlying system environment. The reported timing values should therefore be understood primarily as comparative results within the chosen setup.

Measurement scope. The evaluation focuses on selected technical indicators and representative application scenarios. This provides a sufficient basis for assessing the main effects of the implementation, but it does not cover all aspects of runtime behavior or all possible user conditions in equal depth. In particular, controlled measurements cannot fully substitute observations from productive long-term operation under real usage conditions.

Project context. The implemented solution was developed specifically for QDAcity and reflects its architecture, routing structure, and localization workflow. Although the general design principles are transferable, the concrete results are tied to this project context. Other applications may require different trade-offs depending on their technical and organizational constraints.

Summary. Overall, these limitations define the scope within which the evaluation results should be interpreted. Within this scope, the results provide sufficient evidence that the implemented approach improves maintainability, translation governance, and build-time behavior for the QDAcity application.

6.5 Summary

Overall, the evaluation shows that the implemented approach delivers its clearest benefits in build-time efficiency, translation payload reduction, and operational control. Runtime effects on already fast public pages remain limited, but the route-aligned localization approach improves startup behavior where translation loading is on the critical path. In addition, the evaluated scenarios confirm that the publication-aware localization model behaves consistently across routing, direct access, and both SSG and CSR delivery modes.

7 Conclusions

7.1 Main Contributions

This thesis addressed the problem of scalable content delivery in a modern localized web application. In the baseline architecture of QDAcity, crawler-based prerendering increased build-time cost and operational fragility as route and language coverage grew, while monolithic translation catalogs coupled runtime payload to the total size of the application and made incremental rollout of new languages difficult. The objective of this thesis was therefore to design and implement a more maintainable content delivery architecture that improves controllability, reduces unnecessary runtime work, and supports continuous localization without requiring complete language coverage upfront.

The implemented solution combines four main elements: a Node.js-based SSG pipeline for eligible routes, a route-aligned namespace architecture for localization, CI-supported integration of Weblate into the development workflow, and an incremental translation model with explicit publication states and governed fallback behavior. Together, these changes replace implicit and weakly controlled behavior with explicit contracts at routing, build, and localization boundaries.

The evaluation shows that the clearest gains of the approach lie in build-time efficiency and operational control. In the evaluated setup, the new SSG reduces prerender duration substantially compared with the baseline and provides a more deterministic and maintainable build process. At runtime, the results were more differentiated: already fast public pages showed only limited user-visible changes, whereas the authenticated CSR shell benefited more clearly from route-aligned translation loading because translation payload was directly on the critical rendering path. In addition, the evaluation confirmed that the publication-aware localization model behaved consistently in the evaluated scenarios across routing, direct access, and both SSG and CSR delivery modes.

Overall, the thesis demonstrates that scalable content delivery in a localized React application can be improved not only through isolated performance optimizations, but through a tighter architectural alignment of prerendering, localization, and

publication logic. The main contribution therefore lies in the design and implementation of a practical architecture that improves maintainability, governance, and delivery behavior in the targeted context of QDAcity.

7.2 Limitations and Future Work

The presented results should be interpreted within the scope of the implemented solution and the chosen evaluation setup. The evaluation was conducted under controlled local conditions and focused on selected technical indicators and representative scenarios. Accordingly, the findings provide a grounded assessment for the QDAcity context, but they do not represent a comprehensive characterization of all runtime conditions or deployment environments.

Although the proposed architecture has been implemented to a substantial extent, further development and improvement are still possible. Some supporting capabilities are not yet fully addressed, particularly with regard to regeneration granularity, observability, and adaptation to different device contexts. These aspects represent natural next steps in the continued evolution of the approach.

Future work should therefore focus on operational refinements of the current architecture. Relevant directions include more selective regeneration strategies in the SSG, improved support for bandwidth-sensitive and touch-based usage contexts, stronger runtime telemetry for localization behavior, and additional controls for long-session cache behavior and namespace-level access restrictions.

Taken together, these limitations do not weaken the central result of the thesis. Rather, they indicate that the proposed architecture already provides a useful and effective foundation, while leaving clear and practical directions for further improvement.

Overall, the work demonstrates that aligning rendering, localization, and publication concerns at the architectural level provides a scalable foundation for modern multilingual web applications.

Appendices

A IntlProvider Implementation

The following code snippet is the implementation of the `IntlProvider` component, which is responsible for providing internationalization support to the application.

```

  const isSSG = typeof window !== 'undefined' && window.isSSG;
  function tryRequirePrecompiled(lang) {
    let messages = {};
    try {
      require('fs')
        .readdirSync(`${process.cwd()}/build/messages`)
        .filter((file) => file.endsWith(`.${lang}.json`))
        .forEach((file) => {
          const ms = require(`${process.cwd()}/build/messages/${file}`);
          messages = { ...messages, ...ms };
        });
      return messages;
    } catch (err) {
      console.warn(` Missing precompiled messages for ${lang}`);
      console.warn(err);
      return undefined;
    }
  }

  export const IntlProvider = ({ children, defaultLanguage, defaultRegion })
    => {
    const [initialLang, initialRegion] = userLocale;
    const initLanguage = defaultLanguage || initialLang || 'en';
    const initRegion = defaultRegion || initialRegion || 'US';

    // SSG: seed messages synchronously from precompiled JSON; CSR: keep
    previous behavior
    const ssgMessages = isSSG ? tryRequirePrecompiled(initLanguage) :
    undefined;

    const [locale, setLocale] = useState([initLanguage, initRegion]);
    const [messages, setMessages] = useState(() => {
      if (isSSG && ssgMessages) return ssgMessages;
      return window.defaultMessages || {};
    });

    const { active } = useTranslationNamespaces();

    const effectiveNamespaces = React.useMemo(() => {
      return Array.from(active);
    }, [active]);

    const nsKey = React.useMemo(() => effectiveNamespaces.join('|'), [
    effectiveNamespaces]);

    const [language, region] = locale;
    const intlRouteApi = useIntlRouting(language);

    const changeLanguage = (language = 'en', region = null) => {
```

```
47        let cancel = false;
48
49        (async () => {
50          if (!isSupportedLanguage(language)) return;
51
52          let newLanguage = language;
53          let newRegion = region;
54
55          if (newRegion == null) {
56            const inferredRegion = new Intl.Locale(newLanguage).maximize().
        region;
57            if (!inferredRegion) {
58              newLanguage = 'en';
59              newRegion = 'US';
60            } else {
61              newRegion = inferredRegion;
62            }
63          }
64
65          // SSG: synchronous require of precompiled messages; CSR: existing
        async loader
66          let nextMessages;
67          if (isSSG) {
68            nextMessages = tryRequirePrecompiled(newLanguage) || {};
69          } else {
70            try {
71              nextMessages = await loadMessages(newLanguage, effectiveNamespaces
        );
72            } catch {
73              console.warn(`Couldn't load translation messages for ${newLanguage
        }.`);
74              nextMessages = {};
75            }
76          }
77
78          if (!cancel) {
79            setLocale([newLanguage, newRegion]);
80            setMessages({ ...nextMessages });
81          }
82        })();
83
84        return () => (cancel = true);
85      };
86
87      // Keep your effect, but avoid re-fetching on SSG (already loaded)
88      useEffect(() => {
89        if (!isSSG) {
90          return changeLanguage(defaultLanguage || language, defaultRegion ||
        region);
91        }
92      }, [defaultLanguage, defaultRegion, effectiveNamespaces]);
93
94      const contextState = {
95        language: language,
96        region: region,
97        locale: locale.filter((i) => i).join('-'),
98        userLocale,
99        supportedLanguages: localizationConfig.supportedLanguages,
100       supportedRegions: localizationConfig.supportedRegions,
101       changeLanguage,
102       getNameOfRegion: ({ region, targetLanguage = language }) =>
        getNameOfRegion({ region, targetLanguage }),
```

```
103         getNameOfLanguage: ({ lang, targetLanguage = language }) =>
        getNameOfLanguage({ lang, targetLanguage }),
104         ...intlRouteApi,
105       };
106
107       return (
108         <_IntlProvider
109           locale={contextState.locale}
110           messages={messages}
111           textComponent="span" // Default would be `React.Fragment` but
        currently does break compatibility with ATs
112           defaultRichTextElements={DefaultRichTextElements}
113         >
114           <IntlMerger localizationState={contextState}>{children}</IntlMerger>
115         </_IntlProvider>
116       );
117     };
```

B Custom Link Component

The following code snippet is the implementation of a custom Link component that provides additional functionality for handling locale indicators.

```
import React from 'react';
import { Link as RouterLink, LinkProps as RouterLinkProps, To, useLocation,
  useResolvedPath } from 'react-router-dom';
import { NavHashLink as RouterNavHashLink, NavHashLinkProps as
  RouterNavHashLinkProps } from 'react-router-hash-link';
import styled from 'styled-components';
import { useIntl } from 'common/hooks/useIntl';

const LOCALE_RE = /^\/([a-z]{2})(\/|$)/;

function hasExternalScheme(to: To): boolean {
  if (to && typeof to === 'object') return false;
  if (typeof to !== 'string') return false;

  const s = to.trim().toLowerCase();

  if (s.startsWith('http://')) return true;
  if (s.startsWith('https://')) return true;
  if (s.startsWith('//')) return true;
  if (s.startsWith('mailto:')) return true;
  if (s.startsWith('tel:')) return true;

  return false;
}

function getLocaleFromPath(path: string): string {
  const match = path.match(LOCALE_RE);
  return match?.[1] ?? 'en';
}

function resolvePathnameForLocale(to: To, currentPathname: string): string {
  if (to && typeof to === 'object') {
    const pathname = (to as any).pathname as string | undefined;
    return pathname ?? currentPathname;
  }

  const s = (to ?? '').toString().trim();
  if (!s) return currentPathname;

  if (s.startsWith('#') || s.startsWith('?')) return currentPathname;

  if (s.startsWith('/')) return s;

  return `/${s}`.replace(/\/{2,}/g, '/');
}

type LocaleIndicatorOptions = {
  disableLocaleIndicator?: boolean;
  tooltipText?: (fromLocale: string, toLocale: string) => string;
};

function langPart(locale: string | undefined): string {
  return (locale ?? 'en').trim().toLowerCase().split(/[-_]/)[0];
}

function useLocaleIndicator(to: To, opts: LocaleIndicatorOptions) {
```

```
55      const { pathname } = useLocation();
56      const { formatMessage, locale: intlLocale } = useIntl();
57
58      const urlLocale = langPart(getLocaleFromPath(pathname));
59      const appLocale = langPart(intlLocale);
60
61      const inLocaleTransition = urlLocale !== appLocale;
62      to = useResolvedPath(to).pathname;
63      if (hasExternalScheme(to) || opts.disableLocaleIndicator ||
        inLocaleTransition) {
64        return { title: undefined as string | undefined, icon: null as React.
        ReactNode };
65      }
66
67      const targetPathname = resolvePathnameForLocale(to, pathname);
68      const toLocale = langPart(getLocaleFromPath(targetPathname));
69
70      const changesLocale = urlLocale !== toLocale;
71
72      const title = changesLocale
73        ? (opts.tooltipText?.(urlLocale, toLocale) ??
74          formatMessage({ id: 'common.opensInEnglish', defaultMessage: 'Opens in
         English' }))
75        : undefined;
76
77      return { title, icon: changesLocale ? '\u1F310' : null };
78    }
79
80    const InlineWrapper = styled.span`
81      display: inline-flex;
82      align-items: center;
83      gap: 6px;
84    `;
85
86    const IconWrapper = styled.span`
87      display: inline-flex;
88      align-items: center;
89    `;
90
91    function InlineContent({ children, icon }: { children: React.ReactNode; icon
        : React.ReactNode }) {
92      if (!icon) return <>{children}</>;
93      return (
94        <InlineWrapper>
95          {children}
96          <IconWrapper aria-hidden="true">{icon}</IconWrapper>
97        </InlineWrapper>
98      );
99    }
100
101   // ---- Link ----
102
103   export type LinkProps = RouterLinkProps LocaleIndicatorOptions;
104
105   export const Link = React.forwardRef<HTMLAnchorElement, LinkProps>(function
        Link(
106     { to, children, disableLocaleIndicator, tooltipText, ...rest },
107     ref
108   ) {
109     const { title, icon } = useLocaleIndicator(to, { disableLocaleIndicator,
        tooltipText });
110
111     return (
```

```
      <RouterLink ref={ref} to={to} {...rest} title={title}>
        <InlineContent icon={icon}>{children}</InlineContent>
      </RouterLink>
    );
  });

  // ---- NavHashLink ----

  export type NavHashLinkProps = Omit<RouterNavHashLinkProps, 'to' | 'children
    '> &
    LocaleIndicatorOptions {
      to: RouterNavHashLinkProps['to'];
      children?: React.ReactNode;
    };

  export const NavHashLink = React.forwardRef<HTMLAnchorElement,
    NavHashLinkProps>(function NavHashLink(
    { to, children, disableLocaleIndicator, tooltipText, ...rest },
    ref
  ) {
    const { title, icon } = useLocaleIndicator(to, { disableLocaleIndicator,
    tooltipText });

    return (
      <RouterNavHashLink ref={ref as any} to={to} {...rest} title={title}>
        <InlineContent icon={icon}>{children}</InlineContent>
      </RouterNavHashLink>
    );
  });
```

C AutoTranslationDisclaimer Component

The following code snippet is the implementation of the `AutoTranslationDisclaimer` component, which displays a notice when a page is automatically translated.

```
import React, { useMemo, useEffect, useState } from 'react';
import { useLocation } from 'react-router-dom';
import styled from 'styled-components';
import { FormattedMessage } from 'react-intl';

import languages from 'common/Localization/languages.mjs';
import { BtnDefault } from 'common/ButtonsAndLinks/Btn.jsx';
import { generateEquivalentPath, isRouteAutoTranslated } from 'pages/routes.
  mjs';
import { Link } from 'common/components/Link/Link';

const TranslationNoticeBar = styled.div`
  position: sticky;
  top: 0;
  z-index: 20;
  padding: 10px 16px;
  border: 1px solid transparent;
  background-color: ${(props) => props.theme.bgSecondaryLight};
  border-bottom: 1px solid ${(props) => props.theme.borderDefault};
  display: 'flex';
  min-height: 50px;
  height: auto;

  display: flex;
  gap: 12px;
  align-items: center;
  justify-content: space-between;
  flex-wrap: wrap;
`;

const NoticeText = styled.div`
  font-size: 1em;
  font-family: sans-serif;

  flex: 1 1 auto;
  min-width: 0px;
`;

const NoticeActions = styled.div`
  display: flex;
  gap: 15px;

  flex: 0 0 auto;
  flex-wrap: wrap;
`;

export const AutoTranslationDisclaimer: React.FC = () => {
  const location = useLocation();

  const currentLanguage = useMemo<string>(() => {
    const m = location.pathname.match(new RegExp(`^/(${languages.join('|')})
  (/|$)`));
    return (m?.[1] as string | undefined) || 'en';
  }, [location.pathname]);

  const isAutoTranslated = isRouteAutoTranslated({
```

```
      routePath: location.pathname,
      lang: currentLanguage,
    });

    const lsKey = `autoTranslationDisclaimer:never:${currentLanguage}`;
    const ssKey = `autoTranslationDisclaimer:dismiss:${currentLanguage}:${
    location.pathname}`;

    const [hidden, setHidden] = useState<boolean>(false);

    useEffect(() => {
      if (typeof window === 'undefined') return;

      const never = window.localStorage.getItem(lsKey) === 'true';
      const dismissed = window.sessionStorage.getItem(ssKey) === 'true';
      setHidden(never || dismissed);
    }, [lsKey, ssKey]);

    const dismiss = (): void => {
      window.sessionStorage.setItem(ssKey, 'true');
      setHidden(true);
    };

    const dontShowAgain = (): void => {
      window.localStorage.setItem(lsKey, 'true');
      setHidden(true);
    };

    if (hidden || !isAutoTranslated) return null;

    return (
      <TranslationNoticeBar
        role="status"
        aria-live="polite"
        id="auto-translation-disclaimer"
        className="translationDisclaimer"
      >
        <NoticeText>
          <FormattedMessage
            id="common.autoTranslationDisclaimer.message"
            defaultMessage="This page was automatically translated. Some
    wording may be inaccurate."
          />
        </NoticeText>

        <NoticeActions>
          <Link
            to={generateEquivalentPath({ path: location.pathname, lang: 'en',
    explicit: true })}
            id="auto-translation-disclaimer-show-original-link"
            disableLocaleIndicator
          >
            <BtnDefault id="auto-translation-disclaimer-show-original">
              <FormattedMessage
                id="common.autoTranslationDisclaimer.original"
                defaultMessage="Show original"
              />
            </BtnDefault>
          </Link>

          <BtnDefault onClick={dismiss} id="auto-translation-disclaimer-
    dismiss">
```

```
113             <FormattedMessage id="common.autoTranslationDisclaimer.dismiss"
        defaultMessage="Dismiss" />
114           </BtnDefault>
115
116           <BtnDefault onClick={dontShowAgain} id="auto-translation-disclaimer-
        dont-show-again">
117             <FormattedMessage
118               id="common.autoTranslationDisclaimer.dontShowAgain"
119               defaultMessage="'Dont show again"
120             />
121           </BtnDefault>
122         </NoticeActions>
123       </TranslationNoticeBar>
124     );
125   };
```

D Evaluation Data

The following table contains all measurements used for the empirical evaluations (B = Baseline, T = Target).

Category	**V1**	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**V8**	**V9**	**V10**
Build-time B [min]	3.27	3.22	3.22	3.22	3.23	3.22	3.22	3.21	3.22	3.22
Build-time T 1W [s]	12	13	12	12	12	12	12	12	13	13
Build-time T 2W [s]	9.43	9.56	9.67	9.82	9.88	9.85	10	10	11	9.8
Build-time T 4W [s]	6.58	6.73	6.73	7.13	6.97	6.84	7.02	7.13	7.4	7.16
Build-time T 8W [s]	9.77	9.92	9.64	9.7	9.85	9.8	9.93	10	10	9.87
Build-time T HTML Minify [s]	7.34	8.33	7.02	6.79	6.82	6.96	7.01	6.9	6.8	6.74
LCP 4G B'/' [s]	0.3	0.3	0.29	0.29	0.29	0.3	0.3	0.33	0.3	0.3
LCP 4G B '/de/' [s]	0.29	0.3	0.3	0.3	0.29	0.29	0.28	0.3	0.29	0.29
LCP 4G B '/about/' [s]	0.4	0.28	0.27	0.29	0.28	0.29	0.3	0.28	0.27	0.28
LCP 4G B '/de/uber-uns/' [s]	0.28	0.28	0.29	0.29	0.28	0.29	0.28	0.28	0.28	0.27
LCP 4G B '/qda-software/' [s]	0.31	0.3	0.29	0.3	0.3	0.3	0.28	0.31	0.29	0.3
LCP 4G B '/de/qda-software/' [s]	0.3	0.31	0.29	0.3	0.3	0.3	0.28	0.31	0.29	0.3
LCP 4G T '/' [s]	0.3	0.3	0.3	0.29	0.31	0.29	0.29	0.29	0.3	0.29
LCP 4G T '/de/' [s]	0.32	0.31	0.29	0.3	0.3	0.29	0.29	0.3	0.3	0.31
LCP 4G T '/about/' [s]	0.28	0.28	0.29	0.27	0.28	0.28	0.28	0.27	0.28	0.28
LCP 4G T '/de/uber-uns/' [s]	0.28	0.29	0.28	0.28	0.28	0.3	0.27	0.28	0.28	0.35
LCP 4G T '/qda-software/' [s]	0.3	0.3	0.3	0.33	0.3	0.3	0.31	0.3	0.28	0.33
LCP 4G T '/de/qda-software/' [s]	0.32	0.3	0.31	0.29	0.3	0.3	0.3	0.29	0.3	0.3
LCP 3G B '/' [s]	3.48	3.48	3.47	3.5	3.48	3.48	3.5	3.48	3.49	3.5
LCP 3G B '/de/' [s]	3.4	3.42	3.42	3.48	3.42	3.42	3.43	3.41	3.43	3.42
LCP 3G B '/about/' [s]	3.44	3.46	3.4	3.44	3.46	3.45	3.46	3.46	3.46	3.44
LCP 3G B '/de/uber-uns/' [s]	3.42	3.4	3.42	3.41	3.42	3.41	3.41	3.4	3.41	3.42
LCP 3G B '/qda-software/' [s]	3.48	3.47	3.44	3.44	3.44	3.45	3.47	3.46	3.46	3.45
LCP 3G B '/de/qda-software/' [s]	3.4	3.42	3.38	3.39	3.39	3.4	3.42	3.39	3.37	3.39
LCP 3G T '/' [s]	3.54	3.55	3.52	3.55	3.53	3.54	3.53	3.54	3.56	3.57
LCP 3G T '/de/' [s]	3.47	3.48	3.48	3.48	3.46	3.49	3.55	3.55	3.5	3.5
LCP 3G T '/about/' [s]	3.53	3.52	3.54	3.51	3.51	3.5	3.51	3.53	3.51	3.56
LCP 3G T '/de/uber-uns/' [s]	3.48	3.47	3.46	3.47	3.48	3.49	3.46	3.49	3.46	3.46
LCP 3G T '/qda-software/' [s]	3.53	3.57	3.52	3.51	3.56	3.5	3.52	3.59	3.57	3.58
LCP 3G T '/de/qda-software/' [s]	3.46	3.46	3.46	3.52	3.46	3.44	3.47	3.46	3.45	3.5
LCP 4G B Application [s]	3.12	3.06	3.14	3.11	3.1	3.14	3.08	3.15	3.11	3.0
LCP 4G T Application [s]	2.16	2.06	2.1	2.08	2.11	2.07	2.13	2.08	2.12	2.06

References

Baack, S. (2024). A critical analysis of the largest source for generative ai training data: Common crawl, 2199–2208. https://doi.org/10.1145/3630106.3659033

Flexera. (2024). *Flexera 2024 state of the cloud report*. https://www.flexera.de/about-us/press-center/flexera-2024-state-of-the-cloud-managing-spending-top-challenge-de (accessed: 28.12.2025 14:55).

Gartner. (2024). *Gartner forecasts worldwide public cloud end-user spending to total $723 billion in 2025*. https://www.gartner.com/en/newsroom/press-releases/2024-11-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-total-723-billion-dollars-in-2025 (accessed: 28.12.2025 14:58).

Iso/iec/ieee 24765:2017 systems and software engineering — vocabulary [Cited term entries: 3.1704 (functional requirement) and 3.2621 (nonfunctional requirement)]. (2017). ISO/IEC/IEEE. https://www.iso.org/standard/71952.html

Zigisova, J., & Akrap, I. (2024). *Performance*. https://almanac.httparchive.org/en/2024/performance (accessed: 28.12.2025 15:30).